ACM Classification 2012
Mathematics of computing, Theory of computation, Theory of computation → Data structures design and analysis, Computing methodologies

ISBN 978-3-95977-094-1

Published online and open access by

Publication date
December, 2018

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

License
This work is licensed under a Creative Commons Attribution 3.0 Unported license (CC-BY 3.0): http://creativecommons.org/licenses/by/3.0/legalcode.
In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors’ moral rights:
- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.ISAAC.2018.0
LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

Editorial Board
- Luca Aceto (Chair, Gran Sasso Science Institute and Reykjavik University)
- Susanne Albers (TU München)
- Christel Baier (TU Dresden)
- Javier Esparza (TU München)
- Michael Mitzenmacher (Harvard University)
- Madhavan Mukund (Chennai Mathematical Institute)
- Anca Muscholl (University Bordeaux)
- Catuscia Palamidessi (INRIA)
- Raimund Seidel (Saarland University and Schloss Dagstuhl – Leibniz-Zentrum für Informatik)
- Thomas Schwentick (TU Dortmund)
- Reinhard Wilhelm (Saarland University)

ISSN 1868-8969

http://www.dagstuhl.de/lipics
Contents

Preface
Wen-Lian Hsu, Der-Tsai Lee, and Chung-Shou Liao ............................. 0:xi

Program Committee
................................................................................. 0:xiii

External Reviewers
................................................................................. 0:xv–0:xviii

Invited Talks
Going Beyond Traditional Characterizations in the Age of Big Data and Network Sciences
Shang-Hua Teng ................................................................. 1:1–1:1

Approximate Matchings in Massive Graphs via Local Structure
Clifford Stein .................................................................... 2:1–2:1

Regular Papers
Exploiting Sparsity for Bipartite Hamiltonicity
Andreas Björklund ............................................................... 3:1–3:11

Opinion Forming in Erdős–Rényi Random Graph and Expanders
Ahad N. Zehmakan .............................................................. 4:1–4:13

Colouring \((P_5 + P_4)\)-Free Graphs
Tereza Klimošová, Josef Malík, Tomáš Masařík, Jana Novotná, Daniël Paulusma, and Veronika Slívová ................................................................. 5:1–5:13

The Use of a Pruned Modular Decomposition for Maximum Matching Algorithms on Some Graph Classes
Guillaume Ducoffe and Alexandru Popa ...................................... 6:1–6:13

A Novel Algorithm for the All-Best-Swap-Edge Problem on Tree Spanners
Davide Bilò and Kleitos Papadopoulos ........................................ 7:1–7:12

Efficient Enumeration of Dominating Sets for Sparse Graphs
Kazuhiro Kurita, Kunihiro Wasa, Hiroki Arimura, and Takeaki Uno .............. 8:1–8:13

Complexity of Unordered CNF Games
Md Lutfar Rahman and Thomas Watson ......................................... 9:1–9:12

Half-Duplex Communication Complexity
Kenneth Hoover, Russell Impagliazzo, Ivan Mihajlin, and Alexander V. Smal ...... 10:1–10:12

On the Complexity of Stable Fractional Hypergraph Matching
Takashi Ishizuka and Naoyuki Kamiyama ........................................ 11:1–11:12

Deciding the Closure of Inconsistent Rooted Triples Is NP-Complete
Matthew P. Johnson ............................................................. 12:1–12:13
<table>
<thead>
<tr>
<th>Title</th>
<th>Authors</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>An $O(n^2 \log^2 n)$ Time Algorithm for Minmax Regret Minsum Sink on Path Networks</td>
<td>Binay Bhattacharya, Yuya Higashikawa, Tsunehiko Kameda, and Naoki Katoh</td>
<td>14:1–14:13</td>
</tr>
<tr>
<td>Algorithmic Channel Design</td>
<td>Georgia Avarikioti, Yuyi Wang, and Roger Wattenhofer</td>
<td>16:1–16:12</td>
</tr>
<tr>
<td>Counting Connected Subgraphs with Maximum-Degree-Aware Sieving</td>
<td>Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto</td>
<td>17:1–17:12</td>
</tr>
<tr>
<td>Target Set Selection in Dense Graph Classes</td>
<td>Pavel Dvořák, Dušan Knop, and Tomáš Toufar</td>
<td>18:1–18:13</td>
</tr>
<tr>
<td>Data-Compression for Parametrized Counting Problems on Sparse Graphs</td>
<td>Eun Jung Kim, Maria Serna, and Dimitrios M. Thilikos</td>
<td>20:1–20:13</td>
</tr>
<tr>
<td>Planar Maximum Matching: Towards a Parallel Algorithm</td>
<td>Samir Datta, Raghav Kulkarni, Ashish Kumar, and Anish Mukherjee</td>
<td>21:1–21:13</td>
</tr>
<tr>
<td>Beyond-Planarity: Turán-Type Results for Non-Planar Bipartite Graphs</td>
<td>Patrizio Angelini, Michael A. Bekos, Michael Kaufmann, Maximilian Pfister, and Torsten Ueckerdt</td>
<td>28:1–28:13</td>
</tr>
<tr>
<td>A Dichotomy Result for Cyclic-Order Traversing Games</td>
<td>Yen-Ting Chen, Meng-Tsung Tsai, and Shi-Chun Tsai</td>
<td>29:1–29:13</td>
</tr>
</tbody>
</table>
The b-Matching Problem in Distance-Hereditary Graphs and Beyond
   Guillaume Ducoffe and Alexandru Popa ........................................... 30:1–30:13

New Algorithms for Edge Induced König-Egerváry Subgraph Based on
Gallai-Edmonds Decomposition
   Qilong Feng, Guanlan Tan, Senmin Zhu, Bin Fu, and Jianxin Wang ........ 31:1–31:12

Computing Approximate Statistical Discrepancy
   Michael Matheny and Jeff M. Phillips ........................................... 32:1–32:13

Diversity Maximization in Doubling Metrics
   Alfonso Cevallos, Friedrich Eisenbrand, and Sarah Morell ............... 33:1–33:12

On Polynomial Time Constructions of Minimum Height Decision Tree
   Nader H. Bshouty and Waseem Makhoul ....................................... 34:1–34:12

Improved Algorithms for the Shortest Vector Problem and the Closest Vector
Problem in the Infinity Norm
   Divesh Aggarwal and Priyanka Mukhopadhyay ................................ 35:1–35:13

An Adaptive Version of Brandes’ Algorithm for Betweenness Centrality
   Matthias Bentert, Alexander Dittmann, Leon Kellerhals, André Nichterlein, and
   Rolf Niedermeier ................................................................. 36:1–36:13

Algorithms for Coloring Reconfiguration Under Recolorability Constraints

A Cut Tree Representation for Pendant Pairs
   On-Hei S. Lo and Jens M. Schmidt ........................................... 38:1–38:9

Polyline Drawings with Topological Constraints
   Emilio Di Giacomo, Peter Eades, Giuseppe Liotta, Henk Meijer, and
   Fabrizio Montecchiani ............................................................ 39:1–39:13

Almost Optimal Algorithms for Diameter-Optimally Augmenting Trees
   Davide Bilò ............................................................. 40:1–40:13

Approximation Algorithms for Facial Cycles in Planar Embeddings
   Giordano Da Lozzo and Ignaz Rutter ........................................ 41:1–41:13

An Algorithm for the Maximum Weight Strongly Stable Matching Problem
   Adam Kunysz ................................................................. 42:1–42:13

Approximation Algorithm for Vertex Cover with Multiple Covering Constraints
   Eunpyeong Hong and Mong-Jen Kao .......................................... 43:1–43:11

Correlation Clustering Generalized
   David F. Gleich, Nate Veldt, and Anthony Wirth ............................ 44:1–44:13

Partitioning Vectors into Quadruples: Worst-Case Analysis of a Matching-Based
Algorithm
   Annette M.C. Ficker, Thomas Erlebach, Matúš Mihalák, and Frits C. R. Spieksma 45:1–45:12

Coresets for Fuzzy K-Means with Applications
Streaming Algorithms for Planar Convex Hulls
  Martin Farach-Colton, Meng Li, and Meng-Tsung Tsai .......................... 47:1–47:13

Deterministic Treasure Hunt in the Plane with Angular Hints

Competitive Searching for a Line on a Line Arrangement
  Quirijn Bouts, Thom Castermans, Arthur van Goethem, Marc van Kreveld, and Wouter Meulemans .............................................................. 49:1–49:12

Stabbing Pairwise Intersecting Disks by Five Points
  Sariel Har-Peled, Haïm Kaplan, Wolfgang Mulzer, Liam Roditty, Paul Seiferth, Micha Sharir, and Max Willert .................................................. 50:1–50:12

Point Location in Incremental Planar Subdivisions
  Eunjin Oh ........................................................................ 51:1–51:12

Convex Partial Transversals of Planar Regions
  Vahideh Keikha, Mees van de Kerkhof, Marc van Kreveld, Irina Kostitsyna, Maarten Löffler, Frank Staals, Jérôme Urhausen, Jordi L. Vermeulen, and Lionov Wiratma ................................................................. 52:1–52:12

Extending the Centerpoint Theorem to Multiple Points
  Alexander Pilz and Patrick Schnider .............................................. 53:1–53:13

Approximate Query Processing over Static Sets and Sliding Windows
  Ran Ben Basat, Seungbum Jo, Srinivasa Rao Satti, and Shubham Ugare ........... 54:1–54:12

Multi-Finger Binary Search Trees

On Counting Oracles for Path Problems
  Ivona Bezáková and Andrew Searns .................................................. 56:1–56:12

Reconstructing Phylogenetic Tree From Multipartite Quartet System
  Hiroshi Hirai and Yuni Iwamasa ...................................................... 57:1–57:13

Rectilinear Link Diameter and Radius in a Rectilinear Polygonal Domain
  Elena Arseneva, Man-Kwun Chiu, Matias Korman, Aleksandar Markovic, Yoshio Okamoto, Aurélien Ooms, André van Renssen, and Marcel Roeloffzen ...... 58:1–58:13

Minimizing Distance-to-Sight in Polygonal Domains
  Eunjin Oh ........................................................................ 59:1–59:12

Partially Walking a Polygon
  Franz Aurenhammer, Michael Steinkogler, and Rolf Klein ......................... 60:1–60:9

Stabbing Rectangles by Line Segments – How Decomposition Reduces the Shallow-Cell Complexity
  Timothy M. Chan, Thomas C. van Dijk, Krzysztof Fleszar, Joachim Spoerhase, and Alexander Wolff ................................................................. 61:1–61:13

Impatient Online Matching
  Xingwu Liu, Zhida Pan, Yuyi Wang, and Roger Wattenhofer ................. 62:1–62:12
Extensions of Self-Improving Sorters
  Siu-Wing Cheng and Lie Yan .................................................... 63:1–63:12

Online Scheduling of Car-Sharing Requests Between Two Locations with Many
Cars and Flexible Advance Bookings
  Kelin Luo, Thomas Erlebach, and Yinfeng Xu ................................. 64:1–64:13

Packing Returning Secretaries
  Martin Hoefer and Lisa Wilhelmi ............................................... 65:1–65:12

Simple 2\textsuperscript{f} -Color Choice Dictionaries
  Frank Kammer and Andrej Sajenko ............................................. 66:1–66:12

Succinct Data Structures for Chordal Graphs
  J. Ian Munro and Kaiyu Wu ..................................................... 67:1–67:12

Tree Path Majority Data Structures
  Travis Gagie, Meng He, and Gonzalo Navarro ................................. 68:1–68:12

Encoding Two-Dimensional Range Top-k Queries Revisited
  Seungbum Jo and Srinivasa Rao Satti ......................................... 69:1–69:13

Longest Unbordered Factor in Quasilinear Time
  Tomasz Kociumaka, Ritu Kundu, Manal Mohamed, and Solon P. Pissis .... 70:1–70:13

Packing Sporadic Real-Time Tasks on Identical Multiprocessor Systems

A Relaxed FPTAS for Chance-Constrained Knapsack
  Galia Shabtai, Danny Raz, and Yuval Shavitt ................................ 72:1–72:12

Covering Clients with Types and Budgets
  Dimitris Fotakis, Laurent Gourvès, Claire Mathieu, and Abhinav Srivastav .... 73:1–73:12
This volume contains the proceedings of the 29th International Symposium on Algorithms and Computation (ISAAC 2018), held in Jiaoxi, Yilan, Taiwan, December 16–19, 2018. ISAAC is an annual international symposium that covers the very wide range of topics in the field of algorithms and computation. The main purpose of the symposium is to provide a forum for researchers working in algorithms and theory of computation from all over the world.

In response to our call for papers, we received 195 submissions from 37 countries. Each submission was reviewed by at least three Program Committee members, possibly with the assistance of external reviewers. After an extremely rigorous review process and extensive discussion, the Program Committee selected 71 papers. The best paper award was given to “Exploiting Sparseness for Bipartite Hamiltonicity” by Andreas Björklund. Selected from submissions authored by students only, the best student paper award was given to “Opinion Forming in Erdős-Rényi Random Graph and Expanders” by Ahad N. Zehmakan.

In addition to selected papers, the program also included plenary talks by two prominent invited speakers, Shang-Hua Teng, University of Southern California, USA and Clifford Stein, Columbia University, USA.

We thank all the Program Committee members and external reviewers for their professional service and volunteering their time to review the submissions under time constraints. We also thank all authors who submitted papers for consideration, thereby contributing to the high quality of the conference. We would like also to acknowledge our supporting organizations for their assistance and support, in particular Ministry of Science and Technology, Taiwan, National Tsing Hua University, Academia Sinica, Taiwan, and Association for Algorithms and Computation Theory. Finally, we are deeply indebted to the Organizing Committee Co-Chairs, Ho-Lin Chen and Wing-Kai Hon, whose excellent effort and professional service to the community made the conference an unparalleled success.

December, 2018

Wen-Lian Hsu, Der-Tsai Lee and Chung-Shou Liao
Program Committee

Sang Won Bae (Kyonggi University, Korea)
Mark Bun (Princeton University, USA)
Hubert Chan (The University of Hong Kong, Hong Kong)
Danny Chen (University of Notre Dame, USA)
Ho-Lin Chen (National Taiwan University, Taiwan)
Kai-Min Chung (Academia Sinica, Taiwan)
Leah Epstein (University of Haifa, Israel)
MohammadTaghi Hajiaghayi (University of Maryland, USA)
Wing-Kai Hon (National Tsing Hua University, Taiwan)
Seok-Hee Hong (University of Sydney, Australia)
Sun-Yuan Hsieh (National Cheng Kung University, Taiwan)
Wen-Lian Hsu (Academia Sinica, Taiwan, Co-Chair)
Zhiyi Huang (The University of Hong Kong, Hong Kong)
Giuseppe F. Italiano (University of Rome Tor Vergata, Italy)
Naonori Kakimura (Keio University, Japan)
Ralf Klasing (Universite de Bordeaux, France)
Dieter Kratsch (University of Lorraine - Metz, France)
Minming Li (City University of Hong Kong, Hong Kong)
Yi Li (Nanyang Technological University, Singapore)
Chung-Shou Liao (National Tsing Hua University, Taiwan, Co-Chair)
Pinyan Lu (Shanghai University of Finance and Economics, China)
Kazuhisa Makino (Kyoto University, Japan)
Petra Mutzel (Technical University of Dortmund, Germany)
Jesper Nederlof (Technische Universiteit Eindhoven, Netherlands)
Evanthia Papadopoulou (University of Lugano (USI), Switzerland)
Kunsoo Park (Seoul National University, Korea)
Seth Pettie (University of Michigan, USA)
Marcin Pilipczuk (University of Warsaw, Poland)
Kunihiro Sadakane (The University of Tokyo, Japan)
Tetsuo Shibuya (The University of Tokyo, Japan)
Xiaoming Sun (Chinese Academy of Sciences, China)
Shin-ichi Tanigawa (The University of Tokyo, Japan)
Takeshi Tokuyama (Tohoku University, Japan)
Rob van Stee (University of Siegen, Germany)
Dorothea Wagner (Karlsruhe Institute of Technology, Germany)
Haitao Wang (Utah State University, USA)
Gerhard Woeginger (RWTH Aachen University, Germany)
Prudence Wong (The University of Liverpool, UK)
Hsu-Chun Yen (National Taiwan University, Taiwan)
Huacheng Yu (Harvard University, USA)
Christos Zaroliagis (University of Patras, Greece)
Guochuan Zhang (Zhejiang University, China)
List of External Reviewers

Divesh Aggarwal
Hee-Kap Ahn
Eugenio Angriman
Diego Arroyuelo
Shi Bai
János Balogh
Evangelos Bampas
Soheil Behnezhad
Djamal Belazzougui
Rémy Belmonte
Kristóf Bérczi
Sebastian Berndt
Anup Bhattacharya
Binay Bhattacharya
Sujoy Bhore
Davide Bilò
Andreas Björklund
Hans L. Bodlaender
Marthe Bonamy
Vincenzo Bonifaci
Edouard Bonnet
Flavia Bonomo
Nicolas Bousquet
Cornelius Brand
Ulrik Brandes
Andreas Brandstädt
Robert Bredereck
KarlBringmann
Guido Brückner
Valentin Buchhold
Maike Buchin
Laurent Bulteau
Xavier Bultel
Yixin Cao
Florent Capelli
Chun-Hsiang Chan
Timothy Chan
Panagiotis Charalampopoulos
Vincent Chau
Yanlin Chen
Hua Chen
Hwann-Tzong Chen
Lijie Chen
Po-An Chen
Xue Chen
Chih-Hong Cheng
Man Kwun Chiu
Janka Chlebikova
Rezaul Chowdhury
Mahsa Derakhshan
Gabriele Di Stefano
Nishanth Dikkala
Hu Ding
György Dósa
Andre Droschinsky
Ran Duan
Katharina Eggensperger
Thorsten Ehlers
Robert Elsässer
David Eppstein
Thomas Erlebach
Hossein Esfandiari
Chenglin Fan
Alireza Farhadi
Lene Favrholdt
Henning Fernau
Bingkai Lin  
Guohui Lin  
Chih-Hung Liu  
Hsiang-Hsuan Liu  
Jingcheng Liu  
Jinyan Liu  
Quanquan Liu  
Zhenming Liu  
Takanori Maehara  
Lili Mei  
Othon Michail  
Christopher Morris  
Mikhail Moshkov  
Moritz Muehlenthaler  
Wolfgang Mulzer  
Yakov Nekrich  
Aleksandar Nikolov  
Harumichi Nishimura  
Lutz Oettershagen  
Eunjin Oh  
Shunhao Oh  
Yoshio Okamoto  
Dennis Olivetti  
Taku Onodera  
Christian Ortolf  
Yota Otachi  
Kenta Ozeki  
Dominik Pajak  
Debmalya Panigrahi  
Denis Pankratov  
Nikos Parotsidis  
Daniel Paulusma  
Lehilton L. C. Pedrosa  
Alexander Pilz  
Sheung-Hung Poon  
Alexandru Popa  
Youming Qiao  
Marcel Radermacher  
Ramanujan M. S.  
Marcel Roeloffzen  
Andrei Romashchenko  
Pawel Rzążewski  
Mahdi Safarnezhad  
Yuta Sakai  
Hamed Saleh  
Piotr Sankowski  
Jayalal Sarma  
Nicolas Schabanel  
Christian Scheffer  
Lena Schlipf  
Miriam Schloeter  
Chris Schwiegelshohn  
Masoud Seddighin  
Saeed Seddighin  
Igor Semaev  
C. Seshadhri  
Mudassir Shabbir  
Xiaohan Shan  
Michiel Smid  
Tasuku Soma  
Christian Spisla  
Georgios Stamoulis  
Michalis Stefanidakis  
Fabian Stehn  
Thomas Steinke  
Bintao Sun  
Xiaorui Sun  
Akira Suzuki  
Kenjiro Takazawa  
Yuki Takeuchi
Suguru Tamaki
Xuehou Tan
Pingzhong Tang
Zhihao Gavin Tang
Seiichiro Tani
Sharma V. Thankachan
Guojing Tian
Sophie Toulouse
Charalampos Tsourakakis
Torsten Ueckerdt
Seeun William Umboh
André van Renssen
Sofya Vorotnikova
Alexandros Voudouris
Bow-Yaw Wang
Kai Wang
Yuyi Wang
Zhengyu Wang
Roger Wattenhofer
Franziska Wegner
Zhaohui Wei
Angelika Wiegele
Andrew Winslow
Matthias Wolf
Marcin Wrochna
Weiwei Wu
Xiaowei Wu
Mingyu Xiao
Tao Xiao
Bojian Xu
Easton Li Xu
Jinhui Xu
Yao Xu
Jie Xue
Yutaro Yamaguchi
Hadi Yami
Feidiao Yang
Guang Yang
Deshi Ye
Yu Yokoi
Hung-I Yu
Tim Zeitz
Bernd Zey
Chi Hao Zhang
Jia Zhang
Jialin Zhang
Peng Zhang
Qin Zhang
Yong Zhang
Yu Hao Zhang
Zhao Zhang
Da Zhao
Yingchao Zhao
Chao Dong Zheng
Yuan Zhou
Tobias Zündorf
Going Beyond Traditional Characterizations in the Age of Big Data and Network Sciences

Shang-Hua Teng
University of Southern California, Los Angeles, USA

Abstract
What are efficient algorithms? What are network models? Big Data and Network Sciences have fundamentally challenged the traditional polynomial-time characterization of efficiency and the conventional graph-theoretical characterization of networks.

More than ever before, it is not just desirable, but essential, that efficient algorithms should be scalable. In other words, their complexity should be nearly linear or sub-linear with respect to the problem size. Thus, scalability, not just polynomial-time computability, should be elevated as the central complexity notion for characterizing efficient computation.

For a long time, graphs have been widely used for defining the structure of social and information networks. However, real-world network data and phenomena are much richer and more complex than what can be captured by nodes and edges. Network data are multifaceted, and thus network science requires a new theory, going beyond traditional graph theory, to capture the multifaceted data.

In this talk, I discuss some aspects of these challenges. Using basic tasks in network analysis, social influence modeling, and machine learning as examples, I highlight the role of scalable algorithms and axiomatization in shaping our understanding of “effective solution concepts” in data and network sciences, which need to be both mathematically meaningful and algorithmically efficient.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms, Mathematics of computing → Discrete mathematics, Mathematics of computing → Probability and statistics, Information systems → World Wide Web, Information systems → Data mining

Keywords and phrases scalable algorithms, axiomatization, graph sparsification, local algorithms, advanced sampling, big data, network sciences, machine learning, social influence, beyond graph theory

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.1

Category Invited Talk
Approximate Matchings in Massive Graphs via Local Structure

Clifford Stein
Columbia University, New York City, USA

Abstract
Finding a maximum matching is a fundamental algorithmic problem and is fairly well understood in traditional sequential computing models. Some modern applications require that we handle massive graphs and hence we need to consider algorithms in models that do not allow the entire input graph to be held in the memory of one computer, or models in which the graph is evolving over time.

We introduce a new concept called an “Edge Degree Constrained Subgraph (EDCS)”, which is a subgraph that is guaranteed to contain a large matching, and which can be identified via local conditions. We then show how to use an EDCS to find 1.5-approximate matchings in several different models including Map Reduce, streaming and distributed computing. We can also use an EDCS to maintain a 1.5-optimal matching in a dynamic graph.

This work is joint with Sepehr Asadi, Aaron Bernstein, Mohammad Hossein Bateni and Vahab Marrokni.

2012 ACM Subject Classification Theory of computation → Parallel algorithms, Theory of computation → Online algorithms

Keywords and phrases matching, dynamic algorithms, parallel algorithms, approximation algorithms

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.2

Category Invited Talk
Exploiting Sparsity for Bipartite Hamiltonicity

Andreas Björklund
Department of Computer Science, Lund University, Sweden

Abstract
We present a Monte Carlo algorithm that detects the presence of a Hamiltonian cycle in an $n$-vertex undirected bipartite graph of average degree $\delta \geq 3$ almost surely and with no false positives, in $(2 - 2^{1-\delta})^{n/2} \text{poly}(n)$ time using only polynomial space. With the exception of cubic graphs, this is faster than the best previously known algorithms. Our method is a combination of a variant of Björklund’s $2^{n/2} \text{poly}(n)$ time Monte Carlo algorithm for Hamiltonicity detection in bipartite graphs, SICOMP 2014, and a simple fast solution listing algorithm for very sparse CNF-SAT formulas.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases Hamiltonian cycle, bipartite graph

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.3

Funding This work was supported in part by the Swedish Research Council grant VR-2016-03855, “Algebraic Graph Algorithms”.

1 Introduction

Given an $n$-vertex undirected graph $G = (V, E)$, a Hamiltonian cycle is a vertex permutation $(v_1, v_2, \ldots, v_n)$ such that $v_iv_{i+1} \in E$ for all $i$, including also $v_nv_1 \in E$. The algorithmic problem of deciding if a graph has a Hamiltonian cycle was one of the first problems recognised as NP-hard [12]. For general graphs, an $O(1.657^n)$ time algorithm is known [1]. A natural question to ask is if one can do better in sparse graphs, since the average number of entry and exit alternatives for the cycle at a vertex is smaller. Cygan et al. [5] proved a $(2 + \sqrt{2})^{\text{pw}(P_G)} \text{poly}(n)$ time algorithm that detects a Hamiltonian cycle given a path-decomposition $P_G$ of the graph $G$ of width $\text{pw}(P_G)$. In sparse graphs of average degree $\delta$, path-decompositions of width at most $\delta n/11.538$ can be found in polynomial time as proved by Kneis et al. [13]. Hence the combination of these two results gives faster algorithms for Hamiltonicity in sparse undirected graphs when $\delta < 4.73$. However, the techniques in both [5] and [13] do not seem to directly give much faster algorithms when we guarantee that the graph in addition of being sparse is also bipartite. In contrast, there is a much faster $2^{n/2} \text{poly}(n) \subset O(1.415^n)$ time algorithm for general bipartite graphs [1]. In this paper we propose a method to speed up the latter algorithm to get faster algorithms in sparse bipartite graphs. Our main result says

Theorem 1. There is a Monte Carlo algorithm that given an undirected bipartite graph on $n$ vertices and average degree $\delta \geq 3$ detects if it has a Hamiltonian cycle in $\text{poly}(n)$ space and

$$(2 - 2^{1-\delta})^{n/2} \text{poly}(n)$$

time, without false positives and false negatives with probability at most $2^{-n}$.

See Table 1 for some typical running time bases.
3:2 Exploiting Sparseness for Bipartite Hamiltonicity

Table 1 The base \( c \) in the running time bound \( c^n \) of our algorithm for the first small \( \delta \)'s.

<table>
<thead>
<tr>
<th>( \delta )</th>
<th>3</th>
<th>3.25</th>
<th>3.5</th>
<th>3.75</th>
<th>4</th>
<th>4.25</th>
<th>4.5</th>
<th>4.75</th>
<th>5</th>
<th>5.25</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c )</td>
<td>1.3229</td>
<td>1.3378</td>
<td>1.3503</td>
<td>1.3606</td>
<td>1.3765</td>
<td>1.3826</td>
<td>1.3877</td>
<td>1.3919</td>
<td>1.3953</td>
<td></td>
</tr>
</tbody>
</table>

As far as we know, this is the first example of a Hamiltonicity algorithm that operates in less than \( 2^n/2 \) time for sparse bipartite graphs, save for the special case of cubic (3-regular) graphs for which much faster algorithms have been found. However, the techniques used in these cubic graph algorithms do not scale gracefully with the average degree and already for 4-regular graphs our algorithm is much faster than previous ones. Confer the related work section below for more discussion of earlier algorithms. We also note that our algorithm can be modified to compute the parity of the number of Hamiltonian cycles:

**Theorem 2.** There is a Las Vegas algorithm that given an undirected bipartite graph on \( n \) vertices and average degree \( \delta \geq 3 \) computes the parity of the number of Hamiltonian cycles in \( \text{poly}(n) \) space and

\[
(2 - 2^{-\delta})^{n/2} \text{poly}(n)
\]

expected time.

The combination of techniques employed by our algorithms is similar to the overall idea in the algorithm by Björklund and Husfeldt [2] to compute the parity of the number of Hamiltonian cycles, and subsequently the modular counting algorithm for Hamiltonian cycles in Björklund et al. [4]. The idea is as follows: We first devise an algebraic fingerprint for Hamiltonian cycles, i.e., an exponential sum \( S \) with variables on the edges of the graph such that \( S \) evaluates to a non-zero value only if the underlying graph has a Hamiltonian cycle. Furthermore, if it has a Hamiltonian cycle, it evaluates to a non-zero value with large probability under a random assignment to the variables. Next we show that the exponential sum \( S \) can in fact be randomly chosen from a family of exponential sums, all of which evaluate to the same value. In a randomly chosen exponential sum in the family only a small fraction of the exponentially many terms contribute non-zero values in expectation. If we only knew which those terms were, we could evaluate the exponential sum much faster than summing over all terms. To this end, we show that listing a small superset of the terms that evaluate to non-zero values can be done by means of another exponential time algorithm. In our case here we can encode the interesting terms as solutions to a very sparse CNF-SAT formula. These solutions can in turn be listed by a branching algorithm in combination with a perfect matching algorithm. An alternative listing algorithm for the most interesting values \( \delta \leq 5.5 \) can also be done by a dynamic programming algorithm across a path decomposition of the variable/clause incidence graph of the formula.

1.1 Related Work

Several diverse ideas for Hamiltonicity detection in general and sparse graphs have been pursued to get improved worst case running time bounds. We give here a brief list of the results we are aware of.

1.1.1 Dynamic programming across a path decomposition

Cygan et al. [5] prove that one can decide the existence of Hamiltonian cycles in undirected graphs in \( (2 + \sqrt{2})^{pw(P_G)} \text{poly}(n) \) time, where \( pw(P_G) \) is the path-width of any path-decomposition \( P_G \) of \( G \) given as input. It relies on dynamic programming across the path decomposition and hence requires exponential space usage in \( pw(P_G) \).
Using the best known bound on the path-width for graphs of average degree $\delta$ by Kneis et al. [13], which says that one can construct a path-decomposition $P$ with $\text{pw}(P_G) \leq \frac{2\delta}{1 + \sqrt{2}}$, we get an $O(1.0619^{\delta n})$ time bound. This running time is worse than ours for all $\delta \geq 3$. However, in graphs of maximum degree three, another algorithm of Cygan et al. [6] can be accelerated to run in $3^{\text{pw}(G)}\text{poly}(n)$ time. It uses the efficient path decomposition in cubic graphs devised by Fomin and Høie [10], to arrive at an $O(1.201^n)$ time algorithm. We note that the efficient path-decomposition for cubic graphs of Fomin and Høie [10] is in fact used by the path-decomposition of Kneis et al. [13] in combination with branching. We also note that the very fast algorithm for cubic graphs above is the result not only of the efficient path-decomposition bound in cubic graphs, but also the fact that one only needs 3 states per vertex in a bag for cubic graphs as opposed to the (amortised) $2 + \sqrt{2}$ number of states per vertex in the general algorithm of Cygan et al. [5] (Conf Corollary 1.6 in Cygan et al. [6]). Hence, the cubic case is indeed special for this approach.

### 1.1.2 Branching

A very natural approach to Hamiltonicity detection in cubic graphs is to guess which two of the three edges incident to a vertex are used on the cycle and branch. This will in turn diminish the number of alternatives for how the cycle can pass through the three neighbor vertices. There is an $O(1.251^n)$ algorithm for Hamiltonicity decision in graphs of maximum degree three based on carefully analysed branching [11]. It improves slightly over the $O(1.260^n)$ time algorithm by Eppstein [9]. The algorithms in fact work for the Travelling Salesman problem solving for the minimum cost edge-weighted Hamiltonian cycle. Eppstein also provides an $O(1.297^n)$ time algorithm that can list the solutions and hence determine their number. He also exhibits bipartite maximum degree three graphs that has $\Omega(1.260^n)$ Hamiltonian cycles, demonstrating that any algorithm that enumerates the cycles one-by-one must take this long in the worst case.

### 1.1.3 Directed algorithms

For directed bipartite graphs, there is an $O(1.733^n)$ time algorithm [4]. However, it is still open whether there exists an $O(c^n)$ time algorithm for some $c < 2$ for decision in directed graphs. In directed graphs of average degree $\delta$, counting the Hamiltonian cycles can be done in $2^{\Omega(n/\delta^5)}$ time [4]. The speedup is obtained by a fast modular counting algorithm for small prime powers and the Chinese remainder theorem after noting that the Hamiltonian cycles cannot be too many in a sparse graph.

### 1.1.4 Parity algorithms

In addition to the above decision algorithms, we know of an even faster algorithm for the seemingly more difficult problem of computing the parity of the number of Hamiltonian cycles: There is an $O(1.619^n)$ time algorithm computing the parity in directed graphs [2]. For bipartite graphs there is also an $O(1.5^n)$ time algorithm [2], and for bipartite undirected graphs, the $O(1.415^n)$ time decision algorithm in [1] is also capable of computing the parity. Thomason [16] showed that the parity of the number of Hamiltonian cycles through any specific edge in an undirected graph is always even in a graph where every vertex has odd degree. Note that it does not mean that there always are an even number of Hamiltonian cycles in the graph, e.g. $K_4$ has three Hamiltonian cycles. In fact, computing the parity in planar undirected graphs of maximum degree three is $\oplus P$-hard [18].
1.1.5 Sparsity-aware TSP algorithms

For the \( n \)-vertex Travelling Salesman Problem, i.e., in an edge-weighted graph find the Hamiltonian cycle of smallest total weight, there is a \( 2^{n - \Omega(n/2^2\Delta)} \) time algorithm, with \( \Delta \) the maximum degree in the graph [3]. The proof uses the fact that in a dynamic programming across vertex subsets for Hamiltonian cycles one only needs to consider induced subgraphs that have degree at least two at every vertex. An upper bound of these can be found by the use of Shearer’s lemma. There is also a \( 2^{n - \Omega(n/2^2\delta)} \) time algorithm with \( \delta \) the average degree in the graph [7].

2 The Three Parts of our Design

On a high level, our algorithmic design consists of three parts:

1. Defining an algebraic fingerprint for Hamiltonicity with few non-zero terms in expectation.
2. Encoding possibly non-zero terms as solutions to a CNF-SAT formula.
3. Listing the solutions to the formula by a separate algorithm.

We will first describe each of these three parts before we present the algorithm in pseudocode in Section 3 along with its analysis.

2.1 A family of algebraic fingerprints

Let \( G = (U, V, E) \) be a balanced \(|U| = |V| = n/2\) bipartite undirected graph. We will introduce variables for the directed versions of the edges in \( G \). We will next define a polynomial over a field of characteristic two as an exponential sum of determinants, and prove that it can be used to detect the presence of a Hamiltonian cycle in \( G \). The construction and its analysis are very similar to the one for bipartite graphs in [1] with only one major difference. In [1] matrices in which rows and columns represented vertices from one part of the bipartition were used. Here we take an alternative approach with rows representing one part and columns representing the other part, as we feel it is more natural for describing how many terms can vanish in the summation in sparse graphs.

The idea is to define the polynomial so that it will be non-zero only if there is a Hamiltonian cycle in the graph. We will in fact describe an exponential number of exponential sums, all of which evaluate to the same polynomial. These are identified by variables \( a_{i,j} \) for every \( ij \in E \), and will not contribute to the sum. I.e., regardless of what they are set to, in the exponential sum they will cancel each other and the sum will always evaluate to the same value. They are introduced solely to make sure that under a random assignment, the expected number of non-zero terms in the exponential sum is quite small. We will show that in the next section.

Continuing the fingerprint design, we note that every Hamiltonian cycle \( H \subseteq E \) can be oriented in two ways in the sense that starting from any vertex you can choose which of its two neighbors on the cycle to visit next. We will fix a special vertex \( s \in V \) in the graph which we will use to break symmetry with respect to orientations. I.e., every oriented Hamiltonian cycle will be associated with a monomial in the polynomial, and the introduced asymmetry around \( s \) will ensure that different monomials are assigned to the two orientations of any Hamiltonian cycle to avoid that they cancel each other.

For every edge \( ij \in E \) with \( i \neq s \) and \( j \neq s \), we introduce two identical variables \( z_{i,j} = z_{j,i} \), i.e., they are only different names for the same variable, but for every \( is \in E \) we introduce the two different variables \( z_{i,s} \) and \( z_{s,i} \). For \( ij \notin E \), we set \( z_{i,j} = z_{j,i} = 0 \). We define a
polynomial matrix in a third set of variables $x = \{x_v | v \in V\}$, with rows representing vertices from $V$, and columns representing vertices from $U$, as

$$M_{i,j}(a,x,z) = \sum_{k \in V \setminus \{i\}} z_{i,j,k} (a_{j,k} + x_k). \quad (1)$$

We will use

$$\phi(G) = \sum_{x \in \{0,1\}^{n/2}} \det(M(a,x,z)), \quad (2)$$

as a fingerprint of the existence of a Hamiltonian cycle in $G$. I.e., we sum over all $2^{n/2}$ assignments to $x$ with each $x_i \in \{0,1\}$ to obtain a polynomial in the $z$-variables only (monomials with $a$-variables will cancel each other). We will prove that $\phi(G)$ can only be non-zero if $G$ has a Hamiltonian cycle.

▶ **Lemma 3.** Let $H$ be the family of oriented Hamiltonian cycles in $G$, then

$$\phi(G) = \sum_{h \in H} \prod_{uv \in h} z_{u,v}. \quad (3)$$

**Proof.** Consider the Leibnitz expansion of the $n \times n$ matrix determinant in characteristic two,

$$\det(B) = \sum_{\sigma \in S_n} \prod_{i=1}^n B_{i,\sigma(i)}. \quad (4)$$

If we furthermore expand each product of entries of the matrix $M(a,x,z)$ into a sum of products, we have that each term in the Leibnitz expansion of $\det(M(a,x,z))$ is the product of exactly $n/2$ factors, each of which is either $z_{i,j,k} a_{j,k}$ or $z_{i,j,k} x_k$ for some $i,j,k$ with $i \neq k$. This means a term is the product of $n$ $z$-variables and $n/2$ $a$- or $x$-variables. We first note that if such a term does not contain $x_v$ for some $v \in V$, it will be counted an even number of times in (2). Let $Z$ be the set of these omitted vertices $v$, and note that the term will be included both for $x_v = 0$ and $x_v = 1$ for all $v \in Z$ for any fixed assignment to the other variables. It will be counted $2^{|Z|}$ times, an even number for non-empty $Z$, and hence it will cancel in a characteristic two summation. This also means that in a surviving term, i.e., a term that is counted an odd number of times, each $x_v$ for $v \in V$ is included precisely once as there are at most $n/2$ of them in total. Moreover, this means that no surviving term includes an $a$-variable.

Note that a term that includes $x_v$ for all $v \in V$ describes a cycle cover as every double-arc factor is from a unique row (every vertex in $V$ has outdegree 1), from a unique column (every vertex in $U$ is the endpoint of exactly one arc and the start of another), and has a unique $x$-variable (every vertex in $V$ has indegree 1). Moreover, there are no cycles on exactly 2 vertices due to the constraint $i \neq k$ in the summation in (1). Every cycle cover corresponds to some term in the Leibnitz expansion. If a cycle cover has more than one cycle, we can fix the lexicographically first cycle $C$ that does not go through the special vertex $s$, and reverse its orientation to obtain a different cycle cover with the same monomial term. However this cycle cover is the result of another term in the Leibnitz expansion because the reversed cycle has length larger than 2. If we apply the cycle reversal operation again, the original cycle cover is obtained. Hence we have paired up all cycle covers with at least two cycles, proving that these will also cancel each other in a characteristic two summation. We are left with the Hamiltonian cycles, counted in both orientations as claimed. ▶
2.1.1 Limiting the number of contributing terms

The value of $\phi(G)$ in (3) is insensitive to the $a$-variables. No matter what we set them to the result is the same. We will now take advantage of this fact. By choosing a random assignment to the $a$-variables, we will end up with a formula in which many determinant terms for assignments to the $x$-variables in (2) will be trivially zero, and there is no need for us to evaluate them to compute $\phi(G)$.

We say that an assignment $x : V \rightarrow \{0, 1\}$ is possibly contributing if no column of $M(a,x,z)$ is all zeros. That is, $\det(M(a,x,z))$ is not trivially zero for the reason of having a column with no non-empty entries.

We will bound the probability that not too many assignments to $x$ are possibly contributing under a random assignment to the variables $a$. Consider a fixed assignment $x$ and let $\varepsilon_u$ for $u \in U$ be the event that the column representing $u$ in $M(a,x,z)$ is not identically zero under a randomly uniformly chosen $a$. We have from (1) that if $a_{u,v} + x_v = 0$ (mod 2) for all $uv \in E$ for a fixed $u$, then the event $\varepsilon_u$ happens, hence

$$\Pr(\varepsilon_u) = 1 - \Pr\left(\prod_{uv \in E} (1 + a_{u,v} + x_v) = 1 \pmod{2}\right) = 1 - \frac{1}{2^{d_u}},$$

where $d_u$ is the degree of vertex $u$ in $G$. Clearly, the events $\{\varepsilon_u : u \in U\}$ are mutually independent as they depend on different independent $a$-variables, so

$$\Pr\left(\bigcap_{u \in U} \varepsilon_u\right) = \prod_{u \in U} \left(1 - 2^{-d_u}\right). \quad (4)$$

By using Jensen’s inequality for a concave function $\varphi$,

$$\varphi\left(\frac{\sum_{i=1}^{m} x_i}{m}\right) \geq \frac{\sum_{i=1}^{m} \varphi(x_i)}{m},$$

after noting that $\varphi(x) = \log(1 - 2^{-x})$ is concave, we have from (4) that

$$\Pr\left(\bigcap_{u \in U} \varepsilon_u\right) \leq (1 - 2^{-\delta})^{n/2}.$$

Consequently, by the linearity of expectation, the expected number of possibly contributing terms in (2) under a random assignment to the $a$-variables is at most

$$2^{n/2} \Pr\left(\bigcap_{u \in U} \varepsilon_u\right) \leq (2 - 2^{1-\delta})^{n/2}. \quad (5)$$

2.2 Encoding possibly contributing terms as a CNF SAT formula

We will next turn to how we can classify which $x$-assignments are possibly contributing without explicitly constructing all the matrices $M(a,x,z)$.

To encode the possibly contributing assignments, we consider propositional Boolean formulas in conjunctive normal form (CNF-SAT). An instance $I = (W, C)$ consists of a set of variables $W$, and a set of clauses $C$. Each clause in $C$ is a finite set of literals, and a literal is an occurrence of a variable in $W$ that may or may not be negated. For every assignment $a$ we associate a CNF-SAT instance $I(a) = (W_a, C_a)$, where $W_a$ is a set of $n/2$ Boolean variables,
with one variable \( w_v \) for each \( v \in V \) with the interpretation that \( w_v = \text{True} \iff x_v = 1 \). Remember, we have from (1) that the event \( \varepsilon_u \) happens if some \( a_{u,v} + x_v = 1(\mod 2) \) for some \( uv \in E \). Let \( P_u = \{ v : uv \in E, a_{u,v} = 0 \} \) and \( N_u = \{ v : uv \in E, a_{u,v} = 1 \} \). Hence the clause

\[
C_u = \left( \bigvee_{v \in P_u} w_v \right) \lor \left( \bigvee_{v \in N_u} \neg w_v \right),
\]

where \( \neg w_v \) means the negation of \( w_v \), expresses the truth-value of event \( \varepsilon_u \). We equate \( C_u \) with the set \( \{ C_u, u \in U \} \) as the clauses’ conjunction expresses that all events happen. Consequently, every solution to \( I(a) \) represents an assignment \( x : V \to \{0,1\} \) that is possibly contributing. Note that \( I(a) \) has \( n/2 \) variables and \( n/2 \) clauses. We next turn to how to find them efficiently.

### 2.3 Listing possibly contributing terms

We will show that given a CNF-SAT instance \( I \) on \( \ell \) variables and as many clauses, its solutions can be listed fast enough for our application.

**Lemma 4.** The solutions to a CNF-SAT formula on \( \ell \) variables and at most \( \ell \) clauses can be listed by a polynomial space algorithm in time

\[
O(1.619^\ell + \text{poly}(\ell)),
\]

where \( s \) is the number of solutions.

Remark: Note that in our case \( \ell = n/2 \), so the first term amounts to a \( 1.272^n \) running time term that is dominated by the second term as there will be \( (2 - 2^{1-\delta})^{n/2} \) solutions in expectation according to (5), which is larger than \( 1.272^n \) for \( \delta \geq 3 \).

**Proof.** The algorithm is a two-step procedure. First, as long as there is a variable that occurs both as positive and negative literals in the clauses and there are more than two occurrences of them, we use branching on that variable. Second, when no such variables exist, we can use an idea implicit in Tovey [17] to construct a bipartite perfect matching between clauses and variables to see if there is a solution at all. If so, we branch on any vertex and repeat to learn each variable’s value one at a time for each of the assignments.

A partial assignment sets each variable in \( w \) to either True, False, or Undecided. Initially all variables are Undecided. We will gradually turn partial assignments into full assignments that satisfy the original instance. In the first step, we produce a set \( S \) of tuples of partial assignments and CNF-SAT instances resulting from the original instance by removing all clauses satisfied by the partial assignment. These will all have the following property: if a variable occurs both positively and negatively in the clauses, it has precisely two occurrences. The set \( S \) is generated by taking any yet undecided variable that occurs at least three times and both positively and negatively and setting it in turn to both truth values and recursively continuing on the clauses that are still unsatisfied by the partial assignment so far. If we let \( t(\ell) \) be an upper bound on the number of instances generated this way from an original instance with \( \ell \) clauses, we have that

\[
t(\ell) \leq t(\ell - 1) + t(\ell - 2),
\]

since at least one respectively two clauses are satisfied by the two assignments. In combination with \( t(0) = 1 \), we can solve this recurrence as \( t(\ell) \leq 1.619^\ell \). Hence \( |S| \leq 1.619^\ell \).
In the second step, we consider each partial assignment and instance pair in $S$ in turn. To see if the instance has a solution at all, we can first set all undecided variables that occur either only positively or negatively to the value that would satisfy some remaining clauses. After that we are left with a set of variables that occur in precisely two clauses but of opposite polarity. We construct a bipartite graph with one part representing clauses and one representing vertices, and edges between a clause and its variables. If such a bipartite graph has a perfect matching, we know the instance can be satisfied, by assigning to the variables the truth value that would satisfy the clause associated to the variable by the matching. Conversely, if there is a satisfying assignment, we can also find a perfect matching by connecting clauses with the variable that makes them true.

As long as there is a perfect matching, we know there is at least one satisfying assignment. We branch on any yet undecided variable and check again if there is a perfect matching. If not, we backtrack to another branch, but otherwise we continue to a full assignment and output it. This way we can list all satisfying assignments to the current instance in $S$ with polynomial delay, as checking for a perfect matching is a polynomial time task. In fact, in our case checking for a perfect matching is particularly easy as every variable vertex on one side of the bipartition has only two choices. We can imagine another graph with vertices representing clauses, and variables representing edges, with edges between any two clauses that share a variable. Hall’s marriage theorem now yields that a perfect matching in the original clause-variable bipartite graph exists if every connected component in the latter imagined graph has a cycle. This can be checked in linear time. ▷

2.4 An alternative listing algorithm for $\delta < 5.5$

If the average degree is small enough, we can use another way of listing the solutions, albeit using exponential space. We will use the path decomposition construction of Kneis et al. [13] to obtain a path decomposition $P_G$ of width at most $\text{pw}(P_G) = \delta n / 11.538$ in polynomial time. We first take the incidence graph of the CNF-SAT instance, the bipartite graph with vertices for clauses and variables, and edges between a clause and all its variables.

We can next use a path decomposition of the incidence graph to compute the number of satisfying assignments by a dynamic programming algorithm that uses one bit per variable vertex to keep track of its truth value, and one bit per clause vertex to keep track of whether the clause has been satisfied by the variables seen so far in the dynamic programming. The algorithm is analysed in Samer and Szeider [14] for tree-width, but leaving out the join nodes from the analysis gives a $2^{\text{pw}(G)} \text{poly}(n)$ time algorithm.

From the resulting dynamic programming table, we can list the solutions to the CNF-SAT formula with polynomial delay. Since the time needed to compute the dynamic programming across the path-decomposition is smaller than the expected number of solutions to the CNF-SAT instance

$$2^{\delta/11.538} < (2 - 2^{1-\delta})^{1/2},$$

for all $\delta \leq 5.5$, our running time bound follows.

We also note that Kneis et al. [13] presents another slightly larger path-width decomposition $P'_G$ of size $\text{pw}(P'_G) \leq \delta n / 10.434$ that has a particularly nice structure: All bags share a large fraction of the vertices, and the remaining vertices induce a graph of constant bounded path-width. Hence one can get rid of the exponential space requirement with these path-decompositions for our application by guessing the assignment of the common vertices of all bags (i.e., try all of them), and then solve for the solutions by a path-decomposition dynamic programming of constant size in the remaining vertices. This works for all $\delta \leq 4.97$. 
We are ready to describe and analyse the decision algorithm in pseudocode, and thereby prove Theorem 1. The algorithm operates over the field $\mathbb{GF}(2^\kappa)$. For now, it suffices to think of the parameter $\kappa$ as logarithmic in $n$, it will be given a precise value in the next section. The following procedure is called $n$ times, and as soon as a call returns “yes” we report the detection of a Hamiltonian cycle, otherwise we report no Hamiltonian cycles found.

**HamiltonianCycle($G$).**

1. Pick random values $a : U \times V \rightarrow \{0, 1\}$.
2. Use Lemma 4 to construct a list $L$ of solutions to $I(a)$.
3. If in the process more than $3(2 - 2^{1-\delta})^{n/2}$ solutions are found, abort immediately and return “no”.
4. Set $s = 0$.
5. Pick random values $z : V \times U \rightarrow \mathbb{GF}(2^\kappa)$.
6. Set $z_{u,v} = z_{v,u}$ for $u \neq s$.
7. Pick random values $z : \{s\} \times V \rightarrow \mathbb{GF}(2^\kappa)$.
8. Set $z_{u,v} = 0$ for all $uv \neq E$.
9. For each $x \in L$,
10. Evaluate $t = \det(M(a, x, z))$ over $\mathbb{GF}(2^\kappa)$.
11. $s = s + t$ over $\mathbb{GF}(2^\kappa)$.
12. If $s \neq 0$ return “yes” otherwise return “no”.

### Analysis

We first analyse the correctness of the algorithm. We will set $\kappa$ so that the probability of false negatives in a call to HamiltonianCycle($G$) is at most $\frac{1}{2}$. By calling the procedure $n$ times the claimed false negative probability in Theorem 1 follows.

Consider first step 2 of the algorithm. The expected number of solutions is

$$\mathbb{E}(|L|) = (2 - 2^{1-\delta})^{n/2},$$

according to (5). By Markov’s inequality,

$$\Pr (|L| \geq 3 \cdot \mathbb{E}(|L|)) \leq \frac{1}{3}.$$ 

Hence the false negative rate reported at step 3 is at most $\frac{1}{3}$. If the algorithm proceeds past step 3, steps 4–12 computes (2), which according to Lemma 3 is non-zero only if $G$ has a Hamiltonian cycle. Hence there is no chance of false positives. We use the following well-known Lemma to bound the probability of false negatives.

**Lemma 5 ([Demillo–Lipton–Schwartz–Zippel][8, 15]).** Let $p(x_1, x_2, \ldots, x_m)$ be a nonzero multivariate polynomial of total degree $d$ over a field $F$. Pick $r_1, r_2, \ldots, r_m \in F$ uniformly and independently at random, then

$$\Pr (p(r_1, r_2, \ldots, r_m) = 0) \leq \frac{d}{|F|}.$$
In our case the polynomial has degree \( n \) as seen by (3) and because of the asymmetry around \( s \) it is a non-zero polynomial whenever there are Hamiltonian cycles in the graph. We use \( F = \text{GF}(2^\kappa) \) with \( \kappa = \lceil \log 4n \rceil \) in the above Lemma, to get false negative rate at most \( \frac{1}{4} \).

Combining the two sources of false negatives, we get total false negative probability at most

\[
\frac{1}{3} + 2 \cdot \frac{1}{4} = \frac{1}{2},
\]

as claimed.

We next analyse the running time. In step 2 we use the listing algorithm in Lemma 4 (or the alternative from 2.4) to list the solutions but aborting as soon as \( 3(2 - 2^{1-\delta})^{n/2} \) solutions have been found. According to the lemma, this takes \( O(1.619^n + 3(2 - 2^{1-\delta})^{n/2} \text{poly}(n)) \) time, which is dominated by the second term for \( \delta \geq 3 \). Note that basic arithmetic computations over \( \text{GF}(2^\kappa) \) can be done in \( \text{polylog}(n) \) time. The determinant computation at step 10 requires polynomial time in \( n \) by using Gaussian elimination and multiplication of the diagonal elements to retrieve the value of the determinant (note that the sign of a permutation doesn’t matter in characteristic two).

### 3.2 The proof of the parity counting theorem

We finally show how to modify the decision algorithm to obtain Theorem 2. First, to get a Las Vegas algorithm we simply omit bailing out in step 3 of the algorithm if the list of solutions to \( I(a) \) is too long. This gives a list of solution of expected length \( (2 - 2^{1-\delta})^{n/2} \) according to (5). Second, we will replace the random values \( z_{u,v}, z_{s,w} \) for \( u,v \neq s \) by ones if \( uv \in E \), and zeros otherwise. Third, we will run the algorithm several times, once for each pair of distinct neighbors \( v,w \) of \( s \), setting only \( z_{v,s} = z_{s,w} = 1 \) whereas all other variables incident on \( s \), \( z_{s,u} \) and \( z_{u,s} \) are set to zero for every \( u \). This will make sure that we only count the parity of Hamiltonian cycles through \( v,s,w \) and in one orientation. Summing over all pairs of neighbors to \( s \) we obtain the parity of the number of all Hamiltonian cycles.

### References


Opinion Forming in Erdős–Rényi Random Graph and Expanders

Ahad N. Zehmakan
ETH Zurich, Switzerland
abdolahad.noori@inf.ethz.ch

Abstract
Assume for a graph $G = (V, E)$ and an initial configuration, where each node is blue or red, in each discrete-time round all nodes simultaneously update their color to the most frequent color in their neighborhood and a node keeps its color in case of a tie. We study the behavior of this basic process, which is called majority model, on the Erdős–Rényi random graph $G_{n,p}$ and regular expanders. First we consider the behavior of the majority model on $G_{n,p}$ with an initial random configuration, where each node is blue independently with probability $p_b$ and red otherwise. It is shown that in this setting the process goes through a phase transition at the connectivity threshold, namely $\frac{\log n}{n}$. Furthermore, we say a graph $G$ is $\lambda$-expander if the second-largest absolute eigenvalue of its adjacency matrix is $\lambda$. We prove that for a $\Delta$-regular $\lambda$-expander graph if $\frac{\lambda}{\Delta}$ is sufficiently small, then the majority model by starting from $(\frac{1}{2} - \delta)n$ blue nodes (for an arbitrarily small constant $\delta > 0$) results in fully red configuration in sub-logarithmically many rounds. Roughly speaking, this means the majority model is an “efficient” and “fast” density classifier on regular expanders. As a by-product of our results, we show regular Ramanujan graphs are asymptotically optimally immune, that is for an $n$-node $\Delta$-regular Ramanujan graph if the initial number of blue nodes is $s \leq \beta n$, the number of blue nodes in the next round is at most $cs\Delta$ for some constants $c, \beta > 0$. This settles an open problem by Peleg [33].

2012 ACM Subject Classification Theory of computation

Keywords and phrases majority model, random graph, expander graphs, dynamic monopoly, bootstrap percolation

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.4


1 Introduction

A social network, the graph of relationships among a group of individuals, plays a fundamental role as a medium for the spread of information, ideas, and influence among its members. For example, social media such as Facebook, Twitter, and Instagram have served as a crucial tool for communication and information disseminating in today’s life. Recently, studying different social behaviors like how people form their opinion regarding a new product or an election or how the information spreads through a social network have attracted a substantial amount of attention. Many different processes, from bootstrap percolation [4] to rumor spreading [6], have been introduced to model this sort of social phenomena.

A considerable amount of attention has been devoted to the study of the majority-based models, like voter model, majority bootstrap percolation, and majority model. In the majority bootstrap percolation for a given graph and an initial configuration where each node is blue or red, in each round all blue nodes update their color to the most frequent color in their neighborhood and red nodes stay unchanged. The main motivation behind the
majority bootstrap percolation is to model monotone processes like rumor spreading, where a red/blue node corresponds to an informed/uninformed individual and an informed individual will always stay informed of the rumor. However, to analyze non-monotone processes like the diffusion of two competing technologies over a social network or opinion forming in a community, the majority model is considered where each node updates its color to the most frequent color in its neighborhood and keeps it unchanged in case of a tie. The blue/red color, for instance, could stand for positive/negative opinion regarding a reform proposal.

Let us first fix some notations and define the majority model formally. For a graph $G = (V, E)$ and a node $v \in V$, let $N(v) := \{u \in V : \{v, u\} \in E\}$ be the neighborhood of $v$ and $d(v) := |N(v)|$ be the degree of $v$. Furthermore, for a set $S \subseteq V$, we define $N_S(v) := N(v) \cap S$ and $N(S) := \bigcup_{v \in S} N(v)$. For two node sets $S, S' \subseteq V$, define $e(S, S') := |\{(v, u) : v \in S, u \in S', \{v, u\} \in E\}|$. We also write $n$ for the number of nodes in a graph $G = (V, E)$, i.e. $|V|$.

A configuration is a function $C : V \to \{b, r\}$, where $b$ and $r$ represent blue and red. For a set $S \subseteq V$, $C|_S = a$ means $\forall v \in S, C(v) = a$ for color $a \in \{b, r\}$. For a given initial configuration $C_0$, assume $\forall t \geq 1$ and $v \in V$, $C_t(v)$ is equal to the color that occurs most frequently in $v$’s neighborhood in $C_{t-1}$, and in case of a tie $v$ keeps its current color, i.e. $C_t(v) = C_{t-1}(v)$. This deterministic process is called the majority model. For a given initial configuration $C_0$, let $B(t)$ and $R(t)$ for $t \geq 0$ denote the set of blue and red nodes in $C_t$.

Since for a graph $G$ there are $2^n$ possible configurations and the majority model is a deterministic process, by starting from any initial configuration, the process must eventually reach a cycle of configurations. The length of the cycle and the number of rounds the process needs to reach it are respectively called the period and the consensus time of the process. $2^n$ is a trivial upper bound on both the period and the consensus time of the process. However, Goles and Olivos [18] provided the tight upper bound of two on the period of the process, and Poljak and Turzik [34] showed the consensus time is upper-bounded by $O(n^2)$, which is shown to be tight up to some poly-logarithmic factor by Frischknecht, Keller, and Wattenhofer [13].

The majority model has been studied on different classes of graphs, like lattice [16, 36, 38, 15], infinite lattice [10], random regular graphs [17], and infinite trees [22], when the initial configuration is random, meaning each node is independently blue with probability $p_b$ and red otherwise (without loss of generality, we always assume $p_b \leq 1/2$). We are interested in the behavior of the process when the underlying graph is the Erdős–Rényi random graph $G_{n,p}$, where the node set is $[n] = \{1, \ldots, n\}$ and each edge is added with probability $p$ independently. It is worth to mention that several other dynamic processes also have been studied on $G_{n,p}$ for instance rumor spreading by Fountoulakis, Huber, and Panagiotou [11], bootstrap percolation by Coja-Oghlan, Feige, Krivelevich, and Reichman [7], and interacting particle systems by Schoenebeck and Yu [35].

We prove that in the majority model with $p_b \leq \frac{1}{2} - \omega\left(\frac{1}{\sqrt{n}p}\right)$ on $G_{n,p}$ with $(1 + \epsilon)p^* \leq p$ for any constant $\epsilon > 0$ and $p^* = \frac{\log n}{n}$, the process gets fully red in constant number of rounds asymptotically almost surely (for an $n$-node graph $G$ we say an event happens asymptotically almost surely (a.a.s.) if it happens with probability $1 - o(1)$ as $n$ tends to infinity). We also argue the tightness of this result. This explains the experimental observations from [25].

Furthermore, it is shown that in the majority model on $G_{n,p}$ with $p \leq (1 - \epsilon)p^*$ (for any constant $\epsilon > 0$) if $p_b = o(e^{np/n})$, then the process gets fully red but it does not for $p_b = \omega(e^{np/n})$ a.a.s.

Putting the two aforementioned results together implies that the process exhibits a threshold behavior at $p^*$. More precisely, for $p = (1 + \epsilon)p^*$, if the initial density of blue nodes is slightly less than one half, namely $\frac{1}{2} - \omega(1/\sqrt{\log n})$, then the process gets fully red, but for $p = (1 - \epsilon)p^*$, $p_b$ must be very close to zero, namely smaller than $e^{n(1-\epsilon)\log n}/n = \frac{1}{\epsilon^2}$, to
guarantee that it gets fully red a.a.s. Even though the proofs of the above statements require some effort, the main intuition behind this phase transition simply comes from the fact that $p^*$ is the connectivity threshold for $G_{n,p}$, that is $G_{n,p}$ is connected and disconnected a.a.s. respectively for $(1 + \epsilon)p^* \leq p$ and $p \leq (1 - \epsilon)p^*$.

For $(1 + \epsilon)p^* \leq p$ and $p_b \leq \frac{1}{2} - \omega\left(\frac{1}{\sqrt{n}}\right)$ we distinguish two cases of sparse, $p \leq \frac{n^2}{n}$, and dense, $\frac{n^2}{n} < p$ for some small constant $\gamma > 0$. We argue that in the sparse case a very close neighborhood of each node includes only a constant number of cycles a.a.s., meaning it has a tree-like structure. Building on this tree-like structure, we prove that after constantly many rounds the probability of being blue for each node is so small that the union bound over all nodes yields our desired result. For the dense case, we argue in the first round the number of blue nodes decreases to $\frac{n}{2}$ a.a.s. for a large constant $c'$. Then, relying on the high edge density of the graph we show $s \leq \frac{n}{2}$ blue nodes can create at most $s/n^{2}$ blue nodes in the next round; thus the process gets fully red in constantly many rounds.

For $p \leq (1 - \epsilon)p^*$ and $p_b = \omega(e^{np}/n)$, the idea is to show that there exist sufficiently many constant-size components so that initially there is a fully blue and a fully red component a.a.s., which guarantee the coexistence of both colors. For $p_b = o(e^{np}/n)$, we argue the blue density is small enough to show that in at most two rounds all nodes are red a.a.s.

So far we considered the random setting, but one might approach the model from an extremal point of view, which brings up the very well-studied concept of dynamic monopoly. For a graph $G = (V, E)$ and the majority model a set $D \subseteq V$ is a dynamic monopoly, or shortly dynamo, when the following holds: if in some configuration all nodes in $D$ are red (similarly blue) then the process eventually gets fully red (resp. blue), regardless of the colors of the other nodes. Though the concept of dynamo had been studied before, e.g. by Balogh and Pete [3] and Schonmann [37], it was introduced formally by Kempe, Kleinberg, and Tardos [23] and Peleg [31] independently and motivated from two different contexts. The minimum size of a dynamo has been extensively studied on different graph classes, from the $d$-dimensional lattice, motivated from the literature of statistical physics, by Flocchini, Lodi, Luccio, Pagli, and Santoro [9], Balister, Bollobás, Johnson, and Walters [2], and Jeger and Zehmakan [21] to planar graphs by Peleg [32]. As a notable example, although it had been conjectured by Peleg [32] that the minimum size of a dynamo in any $n$-node graph is in $\Omega(\sqrt{n})$, surprisingly Berger [5] proved for any $n \in \mathbb{N}$ there is an $n$-node graph which has a constant-size dynamo, meaning a constant number of red nodes is sufficient to make the whole graph red. We study the minimum size of a dynamo in $G_{n,p}$, and prove it is larger than $(\frac{1}{2} - \sqrt{\gamma/n})n$ a.a.s. for some constant $c > 0$.

As we discussed, in $G_{n,p}$ and above the connectivity threshold if $p_b$ is slightly less than one half then the process reaches fully red configuration and the minimum size of a dynamo is close to $n/2$ a.a.s. This raises the notorious and well-studied problem of density classification. For a given graph $G$, in the density classification problem [14] the task is to find an updating rule so that for whatever initial configuration, the process gets fully red if the number of reds is more than blues initially, and fully blue otherwise. This is a very central problem in the literature of cellular automata and distributed computing since it is a good test case to measure the power of local computations in gathering global information. This problem turned to be hard, in the sense that Land and Belew [24] proved such an updating rule does not exist even when the underlying graph is a cycle. Mustafa and Pekec [29, 30] approached the problem from a different angle and asked for which classes of graphs the majority model, which is probably one of the most natural candidates, classifies the density, and they proved that it is the case for graphs which have at least $n/2$ nodes of degree $n - 1$. These hardness results however did not stop the quest for the best, although imperfect, solutions and different
weaker variants of the problem have been suggested. A natural way of relaxing the problem would be to require any configuration with less than \((\frac{1}{2} - \delta)n\) blue nodes for some small \(\delta > 0\) results in fully red configuration. What are the graphs for which the majority model classifies the density for reasonably small values of \(\delta\)? To address this question, we argue that regularity and expansion are two determining factors.

Expanders are graphs which are highly connected; meaning to disconnect a large part of the graph, one has to sever many edges. A standard algebraic way of characterizing the expansion of a graph \(G\) is to consider the second-largest absolute eigenvalue of its adjacency matrix, which is denoted by \(\lambda(G)\). For a \(\Delta\)-regular graph \(G\), \(\lambda(G) \leq \Delta\) and smaller \(\lambda(G)\) implies better expansion. We show that in the majority model on a \(\Delta\)-regular graph \(G\), any starting configuration satisfying \(|B(0)| \leq (\frac{1}{2} - \delta)n\), for some fixed but arbitrarily small \(\delta > 0\), results in fully red configuration in sub-logarithmically many rounds if \(\lambda(G)/\Delta\) is sufficiently small. In other words, if initially all nodes have the same color (which could correspond to some information) and an adversary is allowed to corrupt the color of \((\frac{1}{2} - \delta)n\) number of nodes, there is a large class of graphs for which if the nodes simply apply the majority rule, they all retrieve the original color in sub-logarithmically many rounds. Roughly speaking, the majority model is an “efficient” and “fast” density classifier on regular expanders.

In a graph \(G = (V, E)\) and the majority model for two sets \(S, S' \subseteq V\), we say \(S\) controls \(S'\) when the following holds: if \(S\) is fully blue (similarly red) in some configuration \(C\), \(S'\) will be fully blue (resp. red) in the next configuration. The main idea of our results is that in regular expander graphs the number of edges between any two node sets \(S, S'\) is almost completely determined by their cardinality. This simple fact implies the number of nodes that a set can control is proportional to its size, meaning a small set of blue nodes cannot make a big part of the graph blue. Applying this argument iteratively and some careful computations lead into the above result on regular expanders. It seems expansion is not only a sufficient condition for such a behavior but also some sort of a necessary condition since otherwise there can exist a small node set \(S\) so that each node in \(S\) has at least half of its neighbors inside \(S\). Thus, if \(S\) is initially blue, it stays blue forever, regardless of other nodes.

Motivated from fault-local mending in distributed systems, where redundant copies of data are kept and the majority rule is applied to overcome the damage caused by failures, Peleg [33] defined the concept of immunity. An \(n\)-node graph \(G\) is \((\alpha, \beta)\)-immune if any node set of size \(s \leq \beta n\) can control at most \(\alpha s\) nodes in the majority model. Immunity and density classification are related in the sense that for an \((\alpha, \beta)\)-immune graph with \(0 < \alpha, \beta < 1\), \(|B(0)| \leq \beta n\) results in fully red configuration in \(O(\log_{1/\alpha} n)\) rounds. For a \(\Delta\)-regular graph and some constant \(\beta > 0\) the best achievable \(\alpha = \frac{\beta}{\Delta} + \frac{2\sqrt{\beta}}{\Delta - 2}\) for some constant \(c_2 > 0\) because \(s\) nodes can occupy the full neighborhood of at least \(\left\lfloor \frac{s}{\Delta} \right\rfloor\) arbitrary nodes. A \(\Delta\)-regular graph is called asymptotically optimally immune if it is \((\frac{\beta}{\Delta}, \beta)\)-immune for some constants \(c_2, \beta > 0\). These graphs are interesting since they prevent a small number of malicious/failed processors to take over a big fraction of the underlying graph. Peleg proved for any \(\Delta > c_1\) for some constant \(c_1\) there exists an asymptotically optimally immune \(\Delta\)-regular graph (actually he left a logarithmic gap, which was closed by Gärtner and Zehmakan [17], recently). These results are existential, but one might be interested in constructing asymptotically optimally immune \(\Delta\)-regular graphs. For \(\Delta \geq \sqrt{n}\), Peleg established explicit construction of such graphs by using symmetric block designs. He also asked “It would be interesting to construct asymptotically optimally immune regular graphs of degrees smaller than \(\sqrt{n}\).” We settle this problem exploiting a large family of Cayley graphs, called Ramanujan graphs.

In Section 2, we study the behavior of the majority model on the random graph \(G_{n,p}\), and then in Section 3 we present our results regarding regular expander graphs and density classification. The uninterested reader might directly jump into Section 3 since the sections are supposed to stand by their own.
2 Erdős–Rényi Random Graph

In this section, we first study the behavior of the majority model on $G_{n,p}$ with an initial random configuration (where each node is independently blue with probability $p_b$ and red otherwise) above the connectivity threshold in Theorem 3 and below it in Theorem 5. As a corollary of these results it is easy to see that the process goes through a phase transition: above the connectivity threshold if $p_b$ is slightly less than $1/2$, the process gets fully red but below it the value of $p_b$ must be very close to $0$ to guarantee that it gets fully red a.a.s. Then in Theorem 6, we prove the minimum size of a dynamo in $G_{n,p}$ is larger than $(\frac{1}{2} - \frac{\epsilon}{\sqrt{n}})n$ a.a.s. for some constant $\epsilon > 0$, that is $(\frac{1}{2} - \frac{\epsilon}{\sqrt{n}})n$ blue nodes cannot make the whole graph blue no matter how they are placed in the graph.

Let us state two variants of the Chernoff bound which we will use several times later.

**Theorem 1** ([8]). Suppose $x_1, \ldots, x_n$ are independent Bernoulli random variables taking values in $\{0, 1\}$ and let $X$ denote their sum, then

\[ \Pr[(1 + \epsilon)\mathbb{E}[X] \leq X] \leq e^{-\frac{(\epsilon^2/2)\mathbb{E}[X]n}{3}} \quad \text{and} \quad \Pr[X \leq (1 - \epsilon')\mathbb{E}[X]] \leq e^{-\frac{\epsilon'^2/2}{3}\mathbb{E}[X]} \quad \text{for} \quad 0 \leq \epsilon' \leq 1 \]

\[ \Pr[(1 + \epsilon)\mathbb{E}[X] \leq X] \leq e^{-\frac{\epsilon^2/2\mathbb{E}[X]}{3}} \quad \text{for} \quad \epsilon' > 1. \]

To prove Theorem 3, we need Lemma 2, which states in $G_{n,p}$ the degree of each node is concentrated around its expectation. This can be proven by simply applying the Chernoff bound (for a formal proof see e.g. [20]).

**Lemma 2.** In $G_{n,p}$, if $p \geq (1 + \epsilon)\frac{\log n}{n}$ for some constant $\epsilon > 0$, then for each node $v$

\[ \Pr[d(v) < \frac{n\epsilon}{n}] = o\left(\frac{1}{n}\right) \quad \text{for some constant} \quad \epsilon'' > 0 \quad \text{(as a function of} \quad \epsilon). \]

The main idea behind the proof of Theorem 3 is to apply the fact that the edges of each node are distributed randomly all over the graph.

**Theorem 3.** In the majority model with $p_b \leq \frac{1}{2} - \frac{\omega}{\sqrt{n}}$ on $G_{n,p}$ with $p \geq (1 + \epsilon)\frac{\log n}{n}$ for $\epsilon > 0$, the process gets fully red in constant number of rounds a.a.s.

**Proof.** We divide the proof into two parts: dense, which is $p \geq \frac{n^2}{m}$, and sparse, which is $p < \frac{n^2}{m}$ for a sufficiently small constant $\gamma > 0$.

**Dense case.** We first show that in one round a.a.s. the number of blue nodes decreases to $n/c'$ for an arbitrarily large constant $c'$. Then, we prove $n/c'$ blue nodes disappear in constant number of rounds, no matter how they are placed in the graph.

We argue that for an arbitrary node $v$, $\Pr[C_1(v) = b] = o(1)$, which implies the expected number of blue nodes in $C_1$ is equal to $o(n)$. By applying Markov’s inequality [8], the number of blue nodes in $C_1$ is less than $n/c'$ a.a.s. for an arbitrarily large constant $c'$. To compute the probability that node $v$ is blue in $C_1$, consider an arbitrary labeling $u_1, \ldots, u_{d(v)}$ of $v$’s neighbors and define Bernoulli random variable $x_i$ for $1 \leq i \leq d(v)$ to be $1$ if and only if $C_0(u_i) = r$. Assume random variable $d_r(v)$ denotes the number of red nodes in $v$’s neighborhood in $C_0$: clearly, $\mathbb{E}[d_r(v)] = \sum_{i=1}^{d(v)} x_i = d(v)(1 - p_b)$. Let $p_b = 1/2 - \delta$ for $\delta = \omega(1/\sqrt{np})$ then by applying the Chernoff bound (Theorem 1 (i)) we have

\[ \Pr[C_1(v) = b] \leq \Pr[d_r(v) \leq d(v)/2] \leq \Pr[d_r(v) \leq (1 - \delta)(\frac{1}{2} + \delta)d(v)] = \Pr[d_r(v) \leq (1 - \delta)\mathbb{E}[d_r(v)]] \leq e^{-\frac{\delta(1/2 + \delta)d(v)}{2}}. \]
Thus, for some positive constant $\epsilon''$, we have $\mathbb{P}[C_1(v) = b | d(v) \geq \frac{np_e}{c'}] \leq e^{-\frac{\epsilon''(np - \sqrt{np})}{c'} + } = e^{-\omega(1)}$, where we used $\delta = (1/\sqrt{mp})$. Now, by applying Lemma 2,

$$
\begin{align*}
\mathbb{P}[\mathbb{P}[C_1(v) = b]] = \mathbb{P}[\mathbb{P}[C_1(v) = b | d(v) \geq \frac{np_e}{c'}] \cdot \mathbb{P}[d(v) \geq \frac{np_e}{c'}] + \\
\mathbb{P}[\mathbb{P}[C_1(v) = b | d(v) < \frac{np_e}{c'}] \cdot \mathbb{P}[d(v) < \frac{np_e}{c'}] \leq e^{-\omega(1)} \cdot 1 + 1 \cdot o(1) = o(1).
\end{align*}
$$

Now, we prove any non-empty node set of size $s \leq n/c'$ controls at most $s/n^{2}$ nodes a.a.s. This implies by starting from $n/c'$ blue nodes (regardless of how they are placed in the graph) the process gets fully red after at most $2/\gamma$ rounds (notice that $2/\gamma$ is a constant). Let $S$ be a set of size $s \leq n/c'$ and $S'$ be a set of size $s' = s/n^{\gamma/2}$. Since $\mathbb{E}[e(S', V \setminus S)] = s'(n-s)p$, by applying the Chernoff bound (Theorem 1 (i)) and using $p \geq \frac{n^2}{\pi}$, $n-s \geq n/2$, and $s' = s/n^{\gamma/2}$ we have

$$
\mathbb{P} \left[ e(S', V \setminus S) \leq (1 - \frac{1}{2}) \mathbb{E}[e(S', V \setminus S)] \right] \leq e^{-\frac{\epsilon''e(\gamma/2)}{s}} = e^{-\omega(1/s)} \leq e^{-\Theta(sn^{2})} 
$$

(1)

Similarly, since $\mathbb{E}[e(S', S)] = s'np$ again by applying the Chernoff bound (Theorem 1 (iii))

$$
\mathbb{P} \left[ e(S', S) \geq 1 + (n/4s - 1) \mathbb{E}[e(S', S)] \right] \leq e^{-\Theta(sn^{2})} 
$$

(2)

Clearly, $\mathbb{P}[S controls S'] \leq \mathbb{P}[e(S', V \setminus S) \leq e(S', S)]$ because if $e(S', V \setminus S) > e(S', S)$ then there is at least one node in $S'$ which has more than half of its neighbors in $V \setminus S$. Furthermore, $(1 + (n/4s - 1))\mathbb{E}[e(S', S)] = \frac{n}{4s} s'np = \frac{n}{4} s'p$ and by using $(n-s) \geq n/2$ we have $(1 - \frac{1}{2})\mathbb{E}[e(S', V \setminus S)] = \frac{1}{2}s'(n-s)p \geq \frac{n}{4} s'p$. Thus by Equations 1 and 2, $\mathbb{P}[S controls S'] \leq 2e^{-\Theta(sn^{\gamma/2})} = e^{-\Theta(sn^{\gamma/2})}$ since $s \geq 1$.

By the union bound, the probability that there exits a set $S$ of size $s \leq n/c'$ which controls a set of size $s/n^{\gamma/2}$ is bounded by

$$
\sum_{s=1}^{n/c'} \frac{n}{s} \frac{s}{n^{\gamma/2}} e^{-\Theta(sn^{\gamma/2})} \leq \sum_{s=1}^{n/c'} n^{2s} e^{-\Theta(sn^{\gamma/2})} \leq \sum_{s=1}^{n/c'} (n^{2} e^{-\Theta(sn^{\gamma/2})})^{s}.
$$

$(n^{2} e^{-\Theta(sn^{\gamma/2})})^{s}$ is maximized for $s = 1$ since $n^{2} e^{-\Theta(sn^{\gamma/2})} < 1$. Thus, the summation is upper-bounded by $\frac{n^{2}}{s} n^{2} e^{-\Theta(sn^{\gamma/2})} = o(1)$ which proves our claim.

**Sparse case.** Let us first present the following proposition, which roughly speaking states that for small values of $p$, the close neighborhood of each node looks like a tree.

**Proposition 4.** In $G_{n,p}$ with $p < \frac{n^2}{\pi}$ for some small constant $\gamma > 0$, a.a.s. there is no node which is in two different cycles of size 3 or 4.

To prove Proposition 4, it suffices to show a.a.s. there exits no subgraph with $4 \leq k \leq 7$ nodes and $k+1$ edges. By the union bound, the probability of having such a subgraph is upper-bounded by $\sum_{k=4}^{7} \binom{n}{k} \left( \frac{1}{(k+1)} \right) \left( \frac{1}{k} \right)^{k+1} \leq \sum_{k=4}^{7} \Theta(n^{k}) \frac{1}{n^{k+1}} = o(1)$, where in the last step we used the fact that $\gamma$ is a sufficiently small constant (for instance $\gamma < 1/8$). This finishes the proof of Proposition 4.

Now building on this tree-like structure and Lemma 2, we prove the probability that an arbitrary node is blue after two rounds of the process is so small that the union bound over all nodes implies the process is fully red a.a.s. Let $v$ be an arbitrary node and label its neighbors from $u_1$ to $u_d(v)$. We want to upper-bound $\mathbb{P}[C_2(v) = b]$. For $1 \leq i \leq d(v)$ let $u_i^1, \ldots, u_i^{d(u_i)-1}$ be the neighbors of $u_i$ except $v$. Define random variable $X_i$ to be the
number of red nodes among $u_1^i, \cdots, u_{d(u_i)}^i$ in $C_0$. We say node $u_i$ is *almost blue* in $C_1$ if $X_i \leq \frac{d(u_i)}{2}$ (notice if a node is blue in $C_1$, it is also almost blue, but not necessarily the other way around). Now, we bound $\Pr[X_i \leq \frac{d(u_i)}{2}]$, which is the probability that $u_i$ is almost blue. Since $\mathbb{E}[X_i] = (d(u_i) - 1)(1 - p_b)$ for $p_b = \frac{1}{2} - \delta$ and $\delta = O\left(\frac{1}{\sqrt{np}}\right)$, by applying the Chernoff bound (Theorem 1 (i)) we have
\[
\Pr[X_i \leq \frac{d(u_i)}{2}] \leq \Pr[X_i \leq (1 - \delta)(\frac{1}{2} + \delta)(d(u_i) - 1)] = \Pr[X_i \leq (1 - \delta)\mathbb{E}[X_i]] \leq e^{-\frac{\delta^2(1/2 + \delta)(d(u_i) - 1)}{2}}.
\]
Thus for any large constant $c''$, $\Pr[X_i \leq \frac{d(u_i)}{2}|d(u_i) \geq \frac{np}{c''}] \leq e^{-\frac{\delta^2(1/2 + \delta)(\frac{np}{c''})}{2}} = o(1)$ by $\delta = O\left(\frac{1}{\sqrt{np}}\right)$. Now by applying Lemma 2, we have
\[
p_i := \Pr[X_i \leq \frac{d(u_i)}{2}] = \Pr[X_i \leq \frac{d(u_i)}{2}|d(u_i) \geq \frac{np}{c''}] \cdot \Pr[d(u_i) \geq \frac{np}{c''}] + \Pr[X_i \leq \frac{d(u_i)}{2}|d(u_i) < \frac{np}{c''}] \cdot \Pr[d(u_i) < \frac{np}{c''}] \leq o(1) \cdot 1 + 1 \cdot o(1) \leq \delta',
\]
for an arbitrarily small constant $\delta' > 0$.

Now, we bound the probability $\Pr[C_2(v) = b]$. Based on Proposition 4, a.a.s. every node, including $v$, is in at most one cycle of length three, say with $u_1$ and $u_2$, and in at most one cycle of length four, say with $u_3$ and $u_4$, and other $u_i$s share no neighbor except $v$ (see Figure 1). Let $Y$ denote the number of nodes among $u_1, \cdots, u_{d(v)}$ which are almost blue in $C_1$. Then, $\Pr[C_2(v) = b] \leq \Pr[Y \geq \frac{d(v)}{2} - 4]$ because for $u_i$ to be blue in $C_1$ it must be almost blue in $C_1$ by definition and for $v$ to be blue in $C_2$ it needs at least $\frac{d(v)}{2} - 4$ blue nodes among $u_1, \cdots, u_{d(v)}$. Notice that being almost blue and being blue are pretty much the same except being almost blue is not a function of the color of node $v$ (we some sort of assume node $v$ is blue in $C_0$ and still the impact of this assumption is small enough to let us get our desired tail bound). This gives us the independence among $p_i$s for $4 \leq i \leq d(u_i) - 1$ (which we apply in the next step) because the only neighbor they share is $v$. Since we upper-bounded $p_i$ by $\delta'$,
\[
\Pr[C_2(v) = b] \leq \Pr[Y \geq \frac{d(v)}{2} - 4] = \sum_{j = \frac{d(v)}{2} - 4}^{d(v) - 4} \binom{d(v) - 4}{j} \delta'^j (1 - \delta')^{d(v) - 4 - j} \leq 2^d(v) \delta'^{d(v) - 4},
\]
which is equal to $\frac{(2\sqrt{\pi}d(v))^{d(v)}}{e^{d(v)}}$. Thus, $\Pr[C_2(v) = b|d(v) \geq \frac{np}{c''}] \leq \frac{(2\sqrt{\pi}d(v))^{np}}{e^{np}}$ which is less than $e^{-2np}$ by selecting $\delta'$ sufficiently small. Furthermore, $e^{-2np} = o\left(\frac{1}{n}\right)$ by $p \geq (1 + \epsilon)\frac{\log n}{n}$. Now by Lemma 2,
\[
\Pr[C_2(v) = b] = \Pr[C_2(v) = b|d(v) \geq \frac{np}{c''}] \cdot \Pr[d(v) \geq \frac{np}{c''}] + \Pr[C_2(v) = b|d(v) < \frac{np}{c''}] \cdot \Pr[d(v) < \frac{np}{c''}] \leq o\left(\frac{1}{n}\right) \cdot 1 + 1 \cdot o\left(\frac{1}{n}\right) = o\left(\frac{1}{n}\right).
\]
The union bound implies a.a.s. there is no blue node in $C_2$.\[\blacksquare\]

Regarding the tightness of the result of Theorem 3, notice that it does not hold if we replace $O\left(\frac{1}{\sqrt{np}}\right)$ with $\frac{1}{\sqrt{np}}$ for any constant $c$. For $p = 1$, which corresponds to the complete graph, if we color each node blue independently with probability $p_b = \frac{1}{2} - \frac{c}{\sqrt{n}}$ and red otherwise for some constant $c > 0$, then by Central Limit Theorem [8] the probability that more than half of the nodes are blue is a positive constant. This implies the process gets fully blue after one round with some positive constant probability.
4:8 Opinion Forming in Erdős–Rényi Random Graph and Expanders

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{neighborhood.png}
\caption{The neighborhood of node $v$.}
\end{figure}

\textbf{Theorem 5.} In the majority model on $G_{n,p}$ with $p \leq (1 - \epsilon) \frac{\log n}{n}$ for $\epsilon > 0$, a.a.s.

(i) $p_b = \omega(e^{np}/n)$ results in the coexistence of both colors

(ii) $p_b = o(e^{np}/n)$ results in fully red configuration.

We present the proof of part (i). For part (ii), the idea is to show that all nodes distinguish that the major color is red by looking at nodes in distance at most two when $p_b$ is sufficiently small, namely $p_b = o(e^{np}/n)$. The formal proof of part (ii) is given in the extended version of the paper.

\textbf{Proof.} Notice that a blue/red isolated node never changes its color in majority model. Thus, it suffices to show for $P_b = \omega(e^{np}/n)$, a.a.s. there is a blue and a red isolated node in the initial configuration. We discuss the blue case and the proof carries on analogously for red.

Let random variable $X$ denote the number of blue isolated nodes in $C_0$. Consider an arbitrary labeling $v_1, \ldots, v_n$ on the nodes and define the Bernoulli random variable $x_i$, for $1 \leq i \leq n$, to be one if and only if node $v_i$ is isolated and blue in $C_0$. Clearly, $X = \sum_{i=1}^{n} x_i$ and $P[x_i = 1] = p_b(1 - p)^{n-1}$. Thus, by linearity of expectation $\mathbb{E}[X] = np_b(1 - p)^{n-1}$. By applying the estimate $1 - x \geq e^{-x}$ for $0 \leq x \leq 1/2$, plugging in $p_b = \omega(e^{np}/n)$, and using the fact that $e^{np} \leq \frac{\log n}{n} \leq e$, we have $\mathbb{E}[X] \geq n \omega(e^{np}) e^{-np - np^2} = \omega(1)$. Now, we argue that $\mathbb{Var}(X) = o(\mathbb{E}[X]^2)$, which then simply by applying Chebyshev’s inequality [8] implies $P[X = 0] \leq \mathbb{Var}(X)/\mathbb{E}[X]^2 = o(1)$. Therefore, a.a.s. there exist a blue and a red isolated node in $C_0$ which result in the coexistence of both colors.

$\mathbb{Var}(X) = \mathbb{E}[X^2] - \mathbb{E}[X]^2 = \sum_{1 \leq i,j \leq n} \mathbb{E}[x_i \cdot x_j] - \mathbb{E}[X]^2 =$

$\sum_{i=1}^{n} \mathbb{E}[x_i^2] + \sum_{1 \leq i \neq j \leq n} \mathbb{E}[x_i \cdot x_j] - \mathbb{E}[X]^2 = \mathbb{E}[X] + \sum_{1 \leq i \neq j \leq n} P[x_i = 1 \land x_j = 1] - \mathbb{E}[X]^2 =$

$\mathbb{E}[X] + n(n - 1)(1 - p)^{2n-3}p_b^2 - \mathbb{E}[X]^2 = \mathbb{E}[X] + \mathbb{E}[X]^2((1 - \frac{1}{n})\frac{1}{1 - p} - 1)$. Since $\mathbb{E}[X] = \omega(1)$, we have $\mathbb{E}[X] = o(\mathbb{E}[X]^2)$. Furthermore by using $p = o(1)$ we have

$$(1 - \frac{1}{n})\frac{1}{1 - p} - 1 = \frac{p}{1 - p} - \frac{1}{n} \cdot \frac{1}{1 - p} = \frac{pm - 1}{n(1 - p)} = o(1).$$

Putting both together we thus conclude that $\mathbb{Var}[X] = o(\mathbb{E}[X]^2)$.

\textbf{Theorem 6.} In $G_{n,p}$ any dynamo is of size at least $(\frac{1}{2} - \frac{\delta}{\sqrt{n}}) n$ for a large constant $c$ a.a.s.

\textbf{Proof.} The main idea of the proof is similar to the dense case in Theorem 3. It suffices to prove that a.a.s. a set of size $s = (\frac{1}{2} - \delta)n$ for $\delta = \frac{\sqrt{n}}{np}$ cannot control a set of the same size. By definition of controlling, this implies no set of size $s$ or smaller can control a set
of size $s$ or larger; consequently, there is no dynamo of size $s$ or smaller. We show that the probability that an arbitrary node set of size $s$ controls a set of the same size is so small that the union bound over all possibilities yields our claim.

Let $S, S'$ be two node sets of size $s$. We want to bound the probability that $S$ controls $S'$. Since $E[e(S', S)] = s^2p$, by applying the Chernoff bound (Theorem 1 (i)) and $\delta^2 = \frac{c^2}{np}$ for a sufficiently large constant $c$, we have

$$P[(1 + \delta)E[e(S', S)] \leq e(S', S)] \leq e^{-\frac{c^2E(e(S', S))^2}{2}} = e^{-\frac{c^2s^2p}{2np}} \leq e^{-2n}.$$ 

Similarly, since $E[e(S', V \setminus S)] = s(n - s)p$,

$$P[e(S', V \setminus S) \leq (1 - \delta)E[e(S', V \setminus S)]] \leq e^{-\frac{c^2(s(n - s))^2p}{2np}} \leq e^{-2n}.$$ 

Furthermore,

$$(1 + \delta)E[e(S', S)] = (1 + \delta)(\frac{1}{2} - \delta)^2n^2p \leq (1 - \delta)(\frac{1}{2} + \delta)\frac{1}{2} \delta n^2 p = (1 - \delta)E[e(S', V \setminus S)].$$

This implies $P[e(S', S) \geq e(S', V \setminus S)] \leq 2e^{-2n}$. Furthermore, $P[S \text{ controls } S'] \leq P[e(S', S) \geq e(S', V \setminus S)]$ because if $e(S', S) < e(S', V \setminus S)$, then there is a node in $S'$ which shares more than half of its neighbors with $V \setminus S$. Therefore, $P[S \text{ controls } S'] \leq 2e^{-2n}$. By the union bound, the probability that there exist sets $S, S'$ of size $s$ such that $S$ controls $S'$ is upper-bounded by $2^{2n}2e^{-2n} = o(1)$, where $2^{2n}$ is an upper bound on the number of possibilities of choosing sets $S$ and $S'$.

\section{Expanders}

Roughly speaking, our main goal in this section is to show that the majority model is an “efficient” and “fast” density classifier on regular expanders. Let us first state Lemma 7, which is our main tool. Recall that for a graph $G$ the second-largest absolute eigenvalue of its adjacency matrix is denoted by $\lambda(G)$ (to lighten the notation we simply write $\lambda$ where $G$ is clear from the context).

\begin{lemma}
(Lemma 2.3 in [19]) In a $\Delta$-regular graph $G = (V, E)$ for any two node sets $S, S' \subseteq V$, $|e(S, S') - \frac{|S||S'|\Delta}{n}| \leq \lambda\sqrt{|S||S'|}$.
\end{lemma}

In the above lemma, the left-hand side is roughly the deviation between the number of edges among $S$ and $S'$ in $G$ and the expected number of edges among $S$ and $S'$ in the random graph $G_{n, \Delta/n}$ on the node set $V$. A small $\lambda$ (i.e., good expansion) implies that this deviation is small, so the graph is nearly random in this sense; in other words, the number of edges between any two node sets is almost completely determined by their cardinality. Intuitively, this implies in the majority model the number of blue nodes that a blue set $B(t)$ can control is small, so the graph is nearly random in this sense; in other words, the number of edges between any two node sets is almost completely determined by their cardinality. We phrase this argument more formally in Lemma 8 and Lemma 9.

\begin{proof}
For each node in $B(t + 1)$, the number of neighbors in $B(t)$ is at least as large as the number of neighbors in $R(t)$, which implies $e(B(t + 1), R(t)) \leq e(B(t + 1), B(t))$. Now, by applying Lemma 7 to both sides of the inequality, we have

$$\frac{|B(t + 1)||R(t)|\Delta}{n} - \lambda\sqrt{|B(t + 1)||R(t)|} \leq \frac{|B(t + 1)||B(t)|\Delta}{n} + \lambda\sqrt{|B(t + 1)||B(t)|}.$$ 

\end{proof}
Opinion Forming in Erdős–Rényi Random Graph and Expanders

Dividing by $\sqrt{|B(t+1)|}$ and re-arranging the terms give

$$\sqrt{|B(t+1)||\langle R(t) \rangle - |B(t)||} \leq \frac{\lambda n}{\Delta} (\sqrt{|B(t)|} + \sqrt{|R(t)|}).$$

Since $|R(t)| - |B(t)| \geq \frac{\lambda}{\Delta^2} n$ and $\sqrt{|B(t)|} + \sqrt{|R(t)|} \leq 2\sqrt{n}$, we have

$$|B(t+1)|\frac{16\lambda^2 n^2}{\Delta^2} \leq \frac{\lambda^2 n^2}{\Delta^2} 4n \Rightarrow |B(t+1)| \leq \frac{n}{4}.$$

**Lemma 9.** In the majority model and $\Delta$-regular graph $G$, $|B(t)| \leq \frac{n}{4}$ implies $|B(t+1)| \leq \frac{n}{16\Delta^2} B(t)$.

**Proof.** Since each node in $B(t+1)$ must have at least $\Delta/2$ neighbors in $B(t)$, we have $\frac{|B(t+1)|\Delta}{2} \leq \epsilon(B(t+1), B(t))$. Applying Lemma 7 to the right side of the inequality gives

$$|B(t+1)|\Delta \leq \frac{|B(t+1)||B(t)|\Delta}{n} + \lambda \sqrt{|B(t+1)||B(t)|} \Rightarrow$$

$$\sqrt{|B(t+1)||1 - \frac{2|B(t)|}{n}} \leq \frac{2\lambda}{\Delta} \sqrt{|B(t)|}.$$

Now, utilizing $\frac{|B(0)|}{n} \leq \frac{1}{4}$ and taking the square of both sides of the equation imply $|B(t+1)| \leq \frac{n}{16\Delta^2} B(t)$.

Putting Lemma 8 and Lemma 9 together immediately provides Theorem 10.

**Theorem 10.** In the majority model and $\Delta$-regular graph $G$, if $|B(0)| \leq (\frac{1}{2} - \frac{\Delta}{\Delta^2}) n$ then the process gets fully red in $O(\log_{\Delta^2}(\frac{\lambda}{\Delta}) n)$ rounds.

**Corollary 11.** In the majority model and $\Delta$-regular graph $G$ with $\lambda(G) = o(\Delta)$, $|B(0)| \leq (\frac{1}{2} - \delta)n$ for an arbitrary constant $\delta > 0$ results in fully red configuration in sub-logarithmically many rounds.

So far we proved our desired density classification property of the majority model on regular expanders. Now, we discuss that combining these results with some prior works yields some very interesting propositions, in particular solving an open problem by Peleg [33].

The random $\Delta$-regular graph $G_n^\Delta$ is the random graph with a uniform distribution over all $\Delta$-regular graphs on $n$ vertices, say $[n]$. It is known [12] that a.a.s. $\lambda(G_n^\Delta) = O(\sqrt{\Delta})$ for $\Delta \geq 3$. Therefore, Theorem 10 implies that in the majority model on $G_n^\Delta$, if $|B(0)| \leq (\frac{1}{2} - \frac{\Delta}{\Delta^2}) n$ for some large constant $c$ then the process gets fully red a.a.s. This result is already known by Gärtner and Zehmakan [17], however with a much more involved proof.

Recall that a graph is $(\alpha, \beta)$-immune if any node set of size $s \leq \beta n$ controls at most $\alpha s$ nodes, and it is asymptotically optimally immune if it is $(\frac{\alpha}{\Delta}, \frac{\beta}{\Delta})$-immune for some constants $c_2, \beta > 0$. As argued in the introduction, by [33, 17] we know that for any $\Delta > c_1$ for some constant $c_1$, there exists an asymptotically optimally immune $\Delta$-regular graph. However, it would be interesting to construct such graphs explicitly. For $\Delta \geq \sqrt{n}$, Peleg [33] established explicit constructions by using structures for symmetric block designs, and he left the case of $\Delta < \sqrt{n}$ as an open problem. We settle this problem by exploiting a large family of regular Cayley graphs, called Ramanujan graphs. A $\Delta$-regular graph $G$ is Ramanujan if $\lambda(G) = \sqrt{2\Delta - 1}$. Ramanujan graphs are “optimal” expanders because Alon and Boppana [1] proved that for a $\Delta$-regular graph $G$, $\lambda(G) \geq \sqrt{2\Delta - 1} - o(1)$. Thus, Lemma 9 implies that for any $\Delta$-regular Ramanujan graph a node set of size $s \leq \frac{\Delta}{4}$ can control at most $\frac{16\lambda^2 s}{\Delta} \geq \frac{16\lambda^2 s}{\Delta^2} \leq \frac{16\lambda^2 s}{\Delta^2} \leq \frac{16\lambda^2 s}{\Delta^2}$ nodes. This means that any $\Delta$-regular Ramanujan graph is $(\frac{1}{4}, \frac{1}{4})$-immune; i.e., it is asymptotically optimally immune.
Theorem 12. All regular Ramanujan graphs are asymptotically optimally immune.

Lubotzky, Phillips, and Sarnak [26] showed that arbitrarily large $\Delta$-regular Ramanujan graphs exist when $\Delta - 1$ is prime, and moreover they can be explicitly constructed (see also [27, 28]). This result plus Theorem 12 answer the aforementioned question by Peleg.

Finally, as we argued regularity and expansion are sufficient properties for efficient density classification, but a natural question arises: are they also necessary? Some certain level of expansion seems to be needed for a graph to show such a density classification behavior under the majority model because otherwise there can exist a relatively small subset $S$ such that each node in $S$ has at least half of its neighbors in $S$; clearly, if $S$ is fully blue initially, it stays blue forever, even though all the remaining nodes are red. Regarding regularity, if the graph is not regular but almost regular, that is the minimum degree and the maximum degree differ by a constant factor, then the same proof ideas provide similar results. However, large degree gaps can lead into the state where a small subset of nodes of large degrees controls a large set of nodes of small degrees, which is in contrast with density classification. All in all, this would be an interesting question to be addressed rigorously in future work.

References

Opinion Forming in Erdős–Rényi Random Graph and Expanders


Colouring \((P_r + P_s)\)-Free Graphs

Tereza Klímošová  
Department of Applied Mathematics, Charles University, Prague, Czech Republic  
tereza@kam.mff.cuni.cz

Josef Malík  
Czech Technical University in Prague, Czech Republic  
malikjo1@fit.cvut.cz

Tomáš Masařík  
Department of Applied Mathematics, Charles University, Prague, Czech Republic  
masarik@kam.mff.cuni.cz

Jana Novotná  
Department of Applied Mathematics, Charles University, Prague, Czech Republic  
janca@kam.mff.cuni.cz

Daniël Paulusma  
Department of Computer Science, Durham University, Durham, UK  
daniel.paulusma@durham.ac.uk

Veronika Slívová  
Computer Science Institute of Charles University, Prague, Czech Republic  
slivova@iuuk.mff.cuni.cz

---

Abstract

The \(k\)-Colouring problem is to decide if the vertices of a graph can be coloured with at most \(k\) colours for a fixed integer \(k\) such that no two adjacent vertices are coloured alike. If each vertex \(u\) must be assigned a colour from a prescribed list \(L(u) \subseteq \{1, \ldots, k\}\), then we obtain the List \(k\)-Colouring problem. A graph \(G\) is \(H\)-free if \(G\) does not contain \(H\) as an induced subgraph. We continue an extensive study into the complexity of these two problems for \(H\)-free graphs.

We prove that List 3-Colouring is polynomial-time solvable for \((P_2 + P_5)\)-free graphs and for \((P_3 + P_4)\)-free graphs. Combining our results with known results yields complete complexity classifications of 3-Colouring and List 3-Colouring on \(H\)-free graphs for all graphs \(H\) up to seven vertices. We also prove that 5-Colouring is NP-complete for \((P_3 + P_5)\)-free graphs.

2012 ACM Subject Classification  Mathematics of computing → Graph theory

Keywords and phrases  vertex colouring, \(H\)-free graph, linear forest

Digital Object Identifier  10.4230/LIPIcs.ISAAC.2018.5

Related Version  A preprint of this article is available on arXiv [22], https://arxiv.org/abs/1804.11091v2.

Funding  T. Masařík, J. Novotná and V. Slívová were supported by the project GAUK 1277018 and the grant SVV–2017–260452. T. Klímošová was supported by the Center of Excellence – ITI, project P202/12/G061 of GA ČR, by the Center for Foundations of Modern Computer Science (Charles Univ. project UNCE/SCI/004), and by the project GAUK 1277018. T. Masařík was also partly supported by the Center of Excellence – ITI, project P202/12/G061 of GA ČR. V. Slívová was partly supported by the project 17-09142S of GA ČR and Charles University project PRIMUS/17/SCI/9. D. Paulusma was supported by the Leverhulme Trust (RPG-2016-258).

Acknowledgements  We thank Karel Král for helpful comments.

© Tereza Klímošová, Josef Malík, Tomáš Masařík, Jana Novotná, Daniël Paulusma, and Veronika Slívová, licensed under Creative Commons License CC-BY


Editors: Wen-Lian Hsu, Der-Tsai Lee, and Chung-Shou Liao; Article No. 5; pp. 5:1–5:13

Leibniz International Proceedings in Informatics

Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
Graph colouring is a popular concept in Computer Science and Mathematics due to a wide range of practical and theoretical applications, as evidenced by numerous surveys and books on graph colouring and many of its variants (see, for example, [5, 14, 21, 24, 28, 30, 33]). Formally, a colouring of a graph $G = (V, E)$ is a mapping $c : V \rightarrow \{1, 2, \ldots \}$ that assigns each vertex $u \in V$ a colour $c(u)$ in such a way that $c(u) \neq c(v)$ whenever $uv \in E$. If $1 \leq c(u) \leq k$, then $c$ is also called a $k$-colouring of $G$ and $G$ is said to be $k$-colourable. The Colouring problem is to decide if a given graph $G$ has a $k$-colouring for some given integer $k$.

It is well known that Colouring is NP-complete even if $k = 3$ [27]. To pinpoint the reason behind the computational hardness of Colouring one may impose restrictions on the input. This led to an extensive study of Colouring for special graph classes, particularly hereditary graph classes. A graph class is hereditary if it is closed under vertex deletion. As this is a natural property, hereditary graph classes capture a very large collection of well-studied graph classes. It is readily seen that a graph class $\mathcal{G}$ is hereditary if and only if $\mathcal{G}$ can be characterized by a unique set $\mathcal{H}_G$ of minimal forbidden induced subgraphs. If $\mathcal{H}_G = \{H\}$, then a graph $G \in \mathcal{G}$ is called $H$-free.

Kráľ, Kratochvíl, Tuza, and Woeginger [23] started a systematic study into the complexity of Colouring on $H$-free graphs for sets $\mathcal{H}$ of size at most 2. They showed polynomial-time solvability if $H$ is an induced subgraph of $P_4$ or $P_1 + P_3$ and NP-completeness for all other graphs $H$. The classification for the case where $\mathcal{H}$ has size 2 is far from finished; see the summary in [14] or an updated partial overview in [11] for further details. Instead of considering sets $\mathcal{H}$ of size 2, we consider $H$-free graphs and follow another well-studied direction, in which the number of colours $k$ is fixed, that is, $k$ no longer belongs to the input.

$k$-Colouring: Given a graph $G$ does there exist a $k$-colouring of $G$?

A $k$-list assignment of $G$ is a function $L$ with domain $V$ such that the list of admissible colours $L(u)$ of each $u \in V$ is a subset of $\{1, 2, \ldots, k\}$. A colouring $c$ respects $L$ if $c(u) \in L(u)$ for every $u \in V$. If $k$ is fixed, then we obtain the following generalization of $k$-Colouring:

List $k$-Colouring: Given a graph $G$ and a $k$-list assignment $L$ does there exist a colouring of $G$ that respects $L$?

For every $k \geq 3$, $k$-Colouring on $H$-free graphs is NP-complete if $H$ contains a cycle [13] or an induced claw [19, 26]. Hence, the case where $H$ is a linear forest (a disjoint union of paths) remains. The situation is far from settled yet, although many partial results are known [2, 3, 4, 6, 7, 8, 9, 10, 15, 18, 20, 25, 29, 31, 34]. Particularly, the case where $H$ is the $t$-vertex path $P_t$ has been well studied. The cases $k = 4$, $t = 7$ and $k = 5$, $t = 6$ are NP-complete [20]. For $k \geq 1$, $t = 5$ [18] and $k = 3$, $t = 7$ [2], even List $k$-Colouring on $P_t$-free graphs is polynomial-time solvable (see also [14]). For a fixed integer $k$, the $k$-Precolouring Extension problem is to decide a given $k$-colouring defined on an induced subgraph of a graph $G$ can be extended to a $k$-colouring of $G$. Recently it was shown in [7, 8] that $4$-Precolouring Extension, and therefore $4$-Colouring, is polynomial-time solvable for $P_t$-free graphs. In contrast, the more general problem List $4$-Colouring is NP-complete for $P_t$-free graphs [15]. See Table 1 for a summary of all these results.

From Table 1 we see that only the cases $k = 3$, $t \geq 8$ are still open, although some partial results are known for $k$-Colouring for the case $k = 3$, $t = 8$ [9]. The situation when $H$ is a disconnected linear forest $\bigcup P_t$ is less clear. It is known that for every $s \geq 1$, List $s$-Colouring is polynomial-time solvable for $sP_3$-free graphs [4, 14]. For every graph $H$,
Table 1 Summary for $P_1$-free graphs.

<table>
<thead>
<tr>
<th>$k$-Colouring</th>
<th>$k$-Precolouring Extension</th>
<th>List $k$-Colouring</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t$</td>
<td>$k=3$</td>
<td>$k=4$</td>
</tr>
<tr>
<td>$t \leq 5$</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>$t = 6$</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>$t = 7$</td>
<td>P</td>
<td>NP-c</td>
</tr>
<tr>
<td>$t \geq 8$</td>
<td>?</td>
<td>NP-c</td>
</tr>
</tbody>
</table>

List 3-Colouring is polynomial-time solvable for $(H + P_1)$-free graphs if it is polynomially solvable for $H$-free graphs [4, 14]. If $H = rP_1 + P_5$ ($r \geq 0$) a stronger result is known.

**Theorem 1** ([10]). For all $k \geq 1, r \geq 0$, List $k$-Colouring is polynomial-time solvable on $(rP_1 + P_3)$-free graphs.

Theorem 1 cannot be extended to larger linear forests $H$, as List 4-Colouring is NP-complete for $P_6$-free graphs [15] and List 5-Colouring is NP-complete for $(P_2 + P_4)$-free graphs [10]. As mentioned, 5-Colouring is known to be NP-complete for $P_6$-free graphs [20], but the existence of integers $k \geq 3$ and $2 \leq r \leq 5$ such that $k$-Colouring is NP-complete for $(P_r + P_5)$-free graphs has not been shown in the literature.

Another way of making progress is to complete a classification by bounding the size of $H$. It follows from the above results and the ones in Table 1 that for a graph $H$ with $|V(H)| \leq 6$, 3-Colouring and List 3-Colouring (and consequently, 3-Precolouring Extension) are polynomial-time solvable on $H$-free graphs if $H$ is a linear forest, and NP-complete otherwise; see also [14]. In [14] it was also shown that, to obtain the same statement for graphs $H$ with $|V(H)| \leq 7$, only the two cases where $H \in \{P_2 + P_5, P_4 + P_3\}$ must be solved.

**Our Results.** In Section 2 we solve the two missing cases listed above.

**Theorem 2.** List 3-Colouring is polynomial-time solvable for $(P_2 + P_5)$-free graphs and for $(P_3 + P_3)$-free graphs.

We prove Theorem 2 as follows. If the graph $G$ of an instance $(G, L)$ of List 3-Colouring is $P_7$-free, then we can use the aforementioned result of Bonomo et al. [2]. Hence we may assume that $G$ contains an induced $P_7$. We consider every possibility of colouring the vertices of this $P_7$ and try to reduce each resulting instance to a polynomial number of smaller instances of 2-Satisfiability. As the latter problem can be solved in polynomial time, the total running time of the algorithm will be polynomial. The crucial proof ingredient is that we partition the set of vertices of $G$ that do not belong to the $P_7$ into subsets of vertices that are of the same distance to the $P_7$. This leads to several “layers” of $G$. We analyse how the vertices of each layer are connected to each other and to vertices of adjacent layers so as to use this information in the design of our algorithm.

Combining Theorem 2 with the aforementioned known results yields the following complexity classifications for graphs $H$ up to seven vertices.

**Corollary 3.** Let $H$ be a graph with $|V(H)| \leq 7$. If $H$ is a linear forest, then List 3-Colouring is polynomial-time solvable for $H$-free graphs; otherwise already 3-Colouring is NP-complete for $H$-free graphs.

In Section 3 we complement Theorem 2 by proving the following result.

**Theorem 4.** 5-Colouring is NP-complete for $(P_3 + P_3)$-free graphs.
Preliminaries

Let $G = (V, E)$ be a graph. For a vertex $v \in V$, we denote its neighbourhood by $N(v) = \{u \mid uv \in E\}$, its closed neighbourhood by $N[v] = N(v) \cup \{v\}$ and its degree by $\deg(v) = |N(v)|$. For a set $S \subseteq V$, we write $N(S) = \bigcup_{v \in S} N(v) \setminus S$ and $N[S] = N(S) \cup S$, and we let $G[S] = (S, \{uv \mid u, v \in S\})$ be the subgraph of $G$ induced by $S$. The contraction of an edge $e = uv$ removes $u$ and $v$ from $G$ and introduces a new vertex which is made adjacent to every vertex in $N(u) \cup N(v)$. The identification of a set $S \subseteq V$ by a vertex $w$ removes all vertices of $S$ from $G$, introduces $w$ as a new vertex and makes $w$ adjacent to every vertex in $N(S)$.

The length of a path is its number of edges. The distance $\text{dist}_G(u, v)$ between two vertices $u$ and $v$ is the length of a shortest path between them in $G$. The distance $\text{dist}_G(u, S)$ between a vertex $u \in V$ and a set $S \subseteq V \setminus \{v\}$ is defined as $\min\{\text{dist}(u, v) \mid v \in S\}$.

For two graphs $G$ and $H$, we use $G + H$ to denote the disjoint union of $G$ and $H$, and we write $rG$ to denote the disjoint union of $r$ copies of $G$. Let $(G, L)$ be an instance of List 3-Colouring. For $S \subseteq V(G)$, we write $L(S) = \bigcup_{u \in S} L(u)$. We let $P_n$ and $K_n$ denote the path and complete graph on $n$ vertices, respectively. The diamond is the graph obtained from $K_4$ after removing an edge. We say that an instance $(G', L')$ is smaller than some other instance $(G, L)$ of List 3-Colouring if either $G'$ is an induced subgraph of $G$ with $|V(G')| < |V(G)|$; or $G' = G$ and $L'(u) \subseteq L(u)$ for each $u \in V(G)$, such that there exists at least one vertex $u^*$ with $L'(u^*) \subset L(u^*)$.

2 The Two Polynomial-Time Results

In this section we show that List 3-Colouring problem is polynomial-time solvable for $(P_2 + P_3)$-free graphs and for $(P_3 + P_4)$-free graphs. As arguments for these two graph classes are overlapping, we prove both cases simultaneously. Our proof uses the following two results.

- **Theorem 5 ([2]).** List 3-Colouring is polynomial-time solvable for $P_7$-free graphs.

- **Theorem 6 ([12]).** The 2-List Colouring problem is linear-time solvable.

Outline of the proof of Theorem 2. Our goal is to reduce, in polynomial time, an instance $(G, L)$ of List 3-Colouring, where $G$ is $(P_2 + P_3)$-free or $(P_3 + P_4)$-free, to a polynomial number of smaller instances of 2-List-Colouring in such a way that $(G, L)$ is a yes-instance if and only if at least one of the new instances is a yes-instance. As for each of the smaller instances, we can apply Theorem 6, the total running time of our algorithm will be polynomial.

If $G$ is $P_7$-free, then we do not have to do the above and may apply Theorem 5 instead. Hence, we assume that $G$ contains an induced $P_7$. We put the vertices of the $P_7$ in a set $N_0$ and define sets $N_i$ ($i \geq 1$) of vertices of the same distance $i$ from $N_0$; we say that the sets $N_i$ are the layers of $G$. We then analyse the structure of these layers using the fact that $G$ is $(P_2 + P_3)$-free or $(P_3 + P_4)$-free. The first phase of our algorithm is about preprocessing $(G, L)$ after colouring the seven vertices of $N_0$ and applying a number of propagation rules. We consider every possible colouring of the vertices of $N_0$. In each branch we may have to deal with vertices $u$ that still have a list $L(u)$ of size 3. We call such vertices active and prove that they all belong to $N_2$. We then enter the second phase of our algorithm. In this phase we show, via some further branching, that $N_1$-neighbours of active vertices either all have a list from $\{\{h, i\}, \{h, j\}\}$, where $\{h, i, j\} = \{1, 2, 3\}$, or they all have the same list $\{h, i\}$. In the third phase we reduce, again via some branching, to the situation where only the latter option applies: $N_1$-neighbours of active vertices all have the same list. Then in the
fourth and final phase of our algorithm we know so much structure of the instance that we can reduce to a polynomial number of smaller instances of 2-\textsc{List-Colouring} via a new propagation rule identifying common neighbourhoods of two vertices by a single vertex.

\textbf{Theorem 2 (reposted).} \textsc{List 3-Colouring} is polynomial-time solvable for \((P_2 + P_3)\)-free graphs and for \((P_3 + P_4)\)-free graphs.

\textbf{Proof Sketch.} Due to space limitation we omit the proof for the (more involved) case where \(H = P_3 + P_4\). Hence, let \((G, L)\) be an instance of \textsc{List 3-Colouring}, where \(G = (V, E)\) is a \((P_2 + P_3)\)-free graph. Whenever possible, we base our arguments on \((P_3 + P_3)\)-freeness. Since the problem can be solved component-wise, we may assume that \(G\) is connected. If \(G\) contains a \(K_4\), then \(G\) is not 3-colourable, and thus \((G, L)\) is a no-instance. As we can decide if \(G\) contains a \(K_4\) in \(O(n^4)\) time by brute force, we assume that from now on \(G\) is \(K_4\)-free. By brute force we either deduce in \(O(n^7)\) time that \(G\) is \(P_7\)-free or we find an induced \(P_7\) on vertices \(v_1, \ldots, v_7\) in that order. In the first case we use Theorem 5. It remains to deal with the second case.

\textbf{Definition 7 (Layers).} Let \(N_0 = \{v_1, \ldots, v_7\}\). For \(i \geq 1\), we define \(N_i = \{u \mid \text{dist}(u, N_0) = i\}\). We call the sets \(N_i (i \geq 0)\) the \textit{layers} of \(G\).

In the remainder, we consider \(N_0\) to be a fixed set of vertices. That is, we will update \((G, L)\) by applying a number of propagation rules and doing some (polynomial) branching, but we will never delete the vertices of \(N_0\). This will enable us to exploit the \(H\)-freeness of \(G\).

We show the following two claims about layers (proofs omitted).

\textbf{Claim 8.} \(V = N_0 \cup N_1 \cup N_2 \cup N_3\).

\textbf{Claim 9.} \(G[N_2 \cup N_3]\) is the disjoint union of complete graphs of size at most 3, each containing at least one vertex of \(N_2\) (and thus at most two vertices of \(N_3\)).

We will now introduce a number of propagation rules, which run in polynomial time. We are going to apply these rules on \(G\) exhaustively, that is, until none of the rules can be applied anymore. Note that during this process some vertices of \(G\) may be deleted (due to Rules 2 and 2), but as mentioned we will ensure that we keep the vertices of \(N_0\), while we may update the other sets \(N_i (i \geq 1)\). We say that a propagation rule is \textit{safe} if the new instance is a yes-instance of \textsc{List 3-Colouring} if and only if the original instance is so.

\textbf{Rule 1. (no empty lists)} If \(L(u) = \emptyset\) for some \(u \in V\), then return \textbf{no}.

\textbf{Rule 2. (not only lists of size 2)} If \(|L(u)| \leq 2\) for every \(u \in V\), then apply Theorem 6.

\textbf{Rule 3. (connected graph)} If \(G\) is disconnected, then solve \textsc{List 3-Colouring} on each instance \((D, L_D)\), where \(D\) is a connected component of \(G\) that does not contain \(N_0\) and \(L_D\) is the restriction of \(L\) to \(D\). If \(D\) has no colouring respecting \(L_D\), then return \textbf{no}; otherwise remove the vertices of \(D\) from \(G\).

\textbf{Rule 4. (no coloured vertices)} If \(u \notin N_0\), \(|L(u)| = 1\) and \(L(u) \cap L(v) = \emptyset\) for all \(v \in N(u)\), then remove \(u\) from \(G\).

\textbf{Rule 5. (single colour propagation)} If \(u\) and \(v\) are adjacent, \(|L(u)| = 1\), and \(L(u) \subseteq L(v)\), then set \(L(v) := L(v) \setminus L(u)\).

\textbf{Rule 6. (diamond colour propagation)} If \(u\) and \(v\) are adjacent and share two common neighbours \(x\) and \(y\) with \(L(x) \neq L(y)\), then set \(L(x) := L(x) \cap L(y)\) and \(L(y) := L(x) \cap L(y)\).

\textbf{Rule 7. (twin colour propagation)} If \(u\) and \(v\) are non-adjacent, \(N(u) \subseteq N(v)\), and \(L(v) \subset L(u)\), then set \(L(u) := L(v)\).
Rule 8. (triangle colour propagation) If \( u, v, w \) form a triangle, \(|L(u) \cup L(v)| = 2 \) and \(|L(w)| \geq 2 \), then set \( L(w) := L(w) \setminus (L(u) \cup L(v)) \), so \(|L(w)| \leq 1 \).

Rule 9. (no free colours) If \(|L(u) \setminus L(N(u))| \geq 1 \) and \(|L(u)| \geq 2 \) for some \( u \in V \), then set \( L(u) := \{c\} \) for some \( c \in L(u) \setminus L(N(u)) \).

Rule 10. (no small degrees) If \(|L(u)| > |\text{deg}(u)|\) for some \( u \in V \setminus N_0 \), then remove \( u \) from \( G \).

As mentioned, our algorithm will branch at several stages to create a number of new but smaller instances, such that the original instance is a yes-instance if and only if at least one of the new instances is a yes-instance. Unless we explicitly state otherwise, we implicitly assume that Rules 2–2 are applied exhaustively immediately after we branch (see also Claim 10). If we apply Rule 2 or 2 on a new instance, then a no-answer means that we will discard the branch. So our algorithm will only return a no-answer for the original instance \((G, L)\) if we discarded all branches. On the other hand, if we can apply Rule 2 on some new instance and obtain a yes-answer, then we can extend the obtained colouring to a colouring of the graph due to the rules. We will now state (without proof) Claim 10.

\[ \blacktriangleright \textbf{Claim 10. Rules 2–2 are safe and their exhaustive application takes polynomial time. Moreover, if we have not obtained a yes- or no-answer, then afterwards } G \text{ is a connected } (H, K_4)\text{-free graph, such that } V = N_0 \cup N_1 \cup N_2 \cup N_3 \text{ and } 2 \leq |L(u)| \leq 3 \text{ for every } u \in V \setminus N_0. \]

\[ \textbf{Phase 1. Preprocessing } (G, L) \]

In Phase 1 we will preprocess \((G, L)\) using the above propagation rules. To start off the preprocessing we will branch via colouring the vertices of \( N_0 \) in every possible way. By colouring a vertex \( u \), we mean reducing the list of permissible colours to size exactly one. (When \( L(u) = \{c\} \), we consider vertex coloured by colour \( c \).) Thus, when we colour some vertex \( u \), we always give \( u \) a colour from its list \( L(u) \), moreover, when we colour more than one vertex we will always assign distinct colours to adjacent vertices.

\[ \textbf{Branching 1. } (O(1) \text{ branches}) \]

We now consider all possible combinations of colours that can be assigned to the vertices in \( N_0 \). That is, we branch into at most \( 3^7 \) cases, in which \( v_1, \ldots, v_7 \) each received a colour from their list. We note that each branch leads to a smaller instance and that \((G, L)\) is a yes-instance if and only if at least one of the new instances is a yes-instance. Hence, if we applied Rule 2 in some branch, then we discard the branch. If we applied Rule 2 and obtained a no-answer, then we discard the branch as well. If we obtained a yes-answer, then we are done. Otherwise we continue by considering each remaining branch separately. For each remaining branch, we denote the resulting smaller instance by \((G, L)\) again.

We will now introduce a new rule, namely Rule 2. We apply Rule 2 together with the other rules. That is, we now apply Rules 2–2 exhaustively. However, each time we apply Rule 2 we first ensure that Rules 2–2 have been applied exhaustively.

\[ \textbf{Rule 11. } (N_3\text{-reduction}) \text{ If } u \text{ and } v \text{ are in } N_3 \text{ and are adjacent, then remove } u \text{ and } v \text{ from } G. \]

We state (without proofs) the following claims.

\[ \blacktriangleright \textbf{Claim 11. Rule 2, applied after exhaustive application of Rules 2–2, is safe and takes polynomial time. Moreover, afterwards } G \text{ is a connected } (H, K_4)\text{-free graph, such that } V = N_0 \cup N_1 \cup N_2 \cup N_3 \text{ and } 2 \leq |L(u)| \leq 3 \text{ for every } u \in V \setminus N_0. \]
Claim 12. The set $N_3$ is independent, and moreover, each vertex $u \in N_3$ has $|L(u)| = 2$ and exactly two neighbours in $N_2$ which are adjacent.

The following claim follows immediately from Claims 9 and 12.

Claim 13. Every connected component $D$ of $G[N_2 \cup N_3]$ is a complete graph with either $|D| \leq 2$ and $D \subseteq N_2$, or $|D| = 3$ and $|D \cap N_3| \leq 1$.

The following claim (proof omitted) describes the location of the vertices with a list of size 3.

Claim 14. For every $u \in V$, if $|L(u)| = 3$, then $u \in N_2$.

We will now show how to branch in order to reduce the lists of the vertices $u \in N_2$ with $|L(u)| = 3$ by at least one colour. We formalize this approach in the following definition.

Definition 15 (Active vertices). A vertex $u \in N_2$ and its neighbours in $N_1$ are called active if $|L(u)| = 3$. Let $A$ be the set of all active vertices. Let $A_1 = A \cap N_1$ and $A_2 = A \cap N_2$. We deactivate a vertex $u \in A_2$ if we reduce the list $L(u)$ by at least one colour. We deactivate a vertex $u \in A_1$ by deactivating all its neighbours in $A_2$.

Note that every vertex $w \in A_1$ has $|L(w)| = 2$ by Rule 2 applied on the vertices of $N_0$. Hence, if we reduce $L(w)$ by one colour, all neighbours of $w$ in $A_2$ become deactivated by Rule 2, and $w$ is removed by Rule 2. For $1 \leq i \leq j \leq 7$, we let $A(i, j) \subseteq A_1$ be the set of active neighbours of $v_i$ that are not adjacent to $v_j$ and similarly, we let $A(j, i) \subseteq A_1$ be the set of active neighbours of $v_j$ that are not adjacent to $v_i$.

Phase 2. Reduce the number of distinct sets $A(i,j)$

We will now branch into $O(n^{45})$ smaller instances such that $(G, L)$ is a yes-instance of LIST 3-COLOURING if and only if at least one of these new instances is a yes-instance. Each new instance will have the following property:

($P$) for $1 \leq i \leq j \leq 7$ with $j - i \geq 2$, either $A(i,j) = \emptyset$ or $A(j,i) = \emptyset$.

Branching II. ($O(n^{\left(3 \cdot \left(\binom{7}{2} - 6\right)\right)}) = O(n^{45})$ branches)

Consider two vertices $v_i$ and $v_j$ with $1 \leq i \leq j \leq 7$ and $j - i \geq 2$. Assume without loss of generality that $v_i$ is coloured 3 and that $v_j$ is coloured either 1 or 3. Hence, every $w \in A(i,j)$ has $L(w) = \{1, 2\}$, whereas every $w \in A(j,i)$ has $L(w) = \{2, q\}$ for $q \in \{1, 3\}$. We branch as follows. We consider all possibilities where at most one vertex of $A(i,j)$ receives colour 2 (and all other vertices of $A(i,j)$ receive colour 1) and all possibilities where we choose two vertices from $A(i,j)$ to receive colour 2. This leads to $O(n) + O(n^2) = O(n^3)$ branches. In the branches where at most one vertex of $A(i,j)$ receives colour 2, every vertex of $A(i,j)$ will be deactivated. So Property ($P$) is satisfied for $i$ and $j$.

Now consider the branches where two vertices $x_1, x_2$ of $A(i,j)$ both received colour 2. We update $A(j,i)$ accordingly. In particular, afterwards no vertex in $A(j,i)$ is adjacent to $x_1$ or $x_2$, as 2 is a colour in the list of each vertex of $A(j,i)$. We now do some further branching for those branches where $A(j,i) \neq \emptyset$. We consider the possibility where each vertex of $N(A(j,i)) \cap A_2$ is given the colour of $v_j$ and all possibilities where we choose one vertex in $N(A(j,i)) \cap A_2$ to receive a colour different from the colour of $v_j$ (we consider both options to colour such a vertex). This leads to $O(n)$ branches. In the first branch, every vertex of $A(j,i)$ will be deactivated. So Property ($P$) is satisfied for $i$ and $j$.

Now consider a branch where a vertex $u \in N(A(j,i)) \cap A_2$ receives a colour different from the colour of $v_j$. We will show that also in this case every vertex of $A(j,i)$ will be deactivated.
For contradiction, assume that \( A(j, i) \) contains a vertex \( w \) that is not deactivated after colouring \( u \). As \( u \) was in \( N(A(j, i)) \cap A_2 \), we find that \( u \) had a neighbour \( w' \in A(j, i) \). As \( u \) is coloured with a colour different from the colour of \( v_j \), the size of \( L(w') \) is reduced by one (due to Rule 2). Hence \( w' \) got deactivated after colouring \( u \), and thus \( w' \neq w \). As \( w \) is still active, \( w \) has a neighbour \( u' \in A_2 \). As \( u' \) and \( w \) are still active, \( u' \) and \( w \) are not adjacent to \( w' \) or \( u \). Hence, \( u, w, w', v_j, w' \) induce a \( P_5 \) in \( G \). As \( x_1 \) and \( x_2 \) both received colour 2, we find that \( x_1 \) and \( x_2 \) are not adjacent to each other. Hence, \( x_1, v_i, x_2 \) induce a \( P_5 \) in \( G \). Recall that all vertices of \( A(j, i) \), so also \( u \) and \( w' \), are not adjacent to \( x_1 \) or \( x_2 \). As \( u \) and \( w' \) were still active after colouring \( x_1 \) and \( x_2 \), we find that \( u \) and \( w' \) are not adjacent to \( x_1 \) or \( x_2 \) either. By definition of \( A(j, i) \), \( w \) and \( w' \) are not adjacent to \( v_j \). By definition of \( A(i, j) \), \( x_1 \) and \( x_2 \) are not adjacent to \( v_j \). Moreover, \( v_i \) and \( v_j \) are non-adjacent, as \( j - i \geq 2 \). We conclude that \( G \) contains an induced \( P_3 + P_3 \), namely with vertex set \( \{ x_1, v_i, x_2 \} \cup \{ u, w', v_j, w, u' \} \), a contradiction. Hence, every vertex of \( A(j, i) \) is deactivated. So Property (P) is satisfied for \( i \) and \( j \) also for these branches.

Finally by recursive application of the above procedure for all pairs \( v_i, v_j \) such that \( 1 \leq i \leq j \leq 7 \) and \( j - i \geq 2 \) we get a graph satisfying Property (P).

We now consider each resulting instance from Branching II. We denote such an instance by \((G, L)\) again. Note that vertices from \( N_2 \) may now belong to \( N_3 \), as their neighbours in \( N_1 \) may have been removed due to the branching. The exhaustive application of Rules 2–2 preserves (P) (where we apply Rule 2 only after applying Rules 2–2 exhaustively). Hence \((G, L)\) satisfies (P).

We observe that if two vertices in \( A_1 \) have a different list, then they must be different to adjacent vertices of \( N_0 \). Hence, by Property (P), at most two lists of \{ \{1, 2\}, \{1, 3\}, \{2, 3\} \} can occur as lists of vertices of \( A_1 \). Without loss of generality this leads to two cases: either every vertex of \( A_1 \) has list \{ \{1, 2\} \} or \{ \{1, 3\} \} and both lists occur on \( A_1 \); or every vertex of \( A_1 \) has list \{ \{1, 2\} \} only. In the next phase of our algorithm we reduce, via some further branching, every instance of the first case to a polynomial number of smaller instances of the second case.

**Phase 3. Reduce to the case where vertices of \( A_1 \) have the same list**

Recall that we assume that every vertex of \( A_1 \) has list \{ \{1, 2\} \} or \{ \{1, 3\} \}. In this phase we deal with the case when both types of lists occur in \( A_1 \). We first show, without proof, the following two claims.

**Claim 16.** Let \( i \in \{1, 3, 5, 7\} \). Then every vertex from \( A_1 \cap N(v_i) \) is adjacent to some vertex \( v_j \) with \( j \notin \{i - 1, i, i + 1\} \).

**Claim 17.** It holds that \( N(A_1) \cap N_0 = \{ v_{i-1}, v_i, v_{i+1} \} \) for some \( 2 \leq i \leq 6 \). Moreover, we may assume without loss of generality that \( v_{i-1} \) and \( v_{i+1} \) have colour 3 and both are adjacent to all vertices of \( A_1 \) with list \{ \{1, 2\} \}, whereas \( v_i \) has colour 2 and is adjacent to all vertices of \( A_1 \) with list \{ \{1, 3\} \}.

By Claim 17, we can partition the set \( A_1 \) into two (non-empty) sets \( X_{1,2} \) and \( X_{1,3} \), where \( X_{1,2} \) is the set of vertices in \( A_1 \) with list \{ \{1, 2\} \} whose only neighbours in \( N_0 \) are \( v_{i-1} \) and \( v_{i+1} \) (which both have colour 3) and \( X_{1,3} \) is the set of vertices in \( A_1 \) with list \{ \{1, 3\} \} whose only neighbour in \( N_0 \) is \( v_i \) (which has colour 2).

Our goal is to show that we can branch into at most \( O(n^2) \) smaller instances, in which either \( X_{1,2} = \emptyset \) or \( X_{1,3} = \emptyset \), such that \((G, L)\) is a yes-instance of List 3-Colouring if and only if at least one of these smaller instances is a yes-instance. Then afterwards it suffices to
show how to deal with the case where all vertices in $A_1$ have the same list in polynomial time; this will be done in Phase 4 of the algorithm. We start with the following $O(n)$ branching procedure (in each of the branches we may do some further $O(n)$ branching later on).

**Branching III.** ($O(n)$ branches)

We branch by considering the possibility of giving each vertex in $X_{1,2}$ colour 2 and all possibilities of choosing a vertex in $X_{1,2}$ and giving it colour 1. This leads to $O(n)$ branches. In the first branch we obtain $X_{1,2} = \emptyset$. Hence we can start Phase 4 for this branch. We now consider every branch in which $X_{1,2}$ and $X_{1,3}$ are both nonempty. For each such branch we will create $O(n)$ smaller instances of List 3-Colouring, where $X_{1,3} = \emptyset$, such that $(G, L)$ is a yes-instance of List 3-Colouring if and only if at least one of the new instances is a yes-instance.

Let $w \in X_{1,2}$ be the vertex that was given colour 1 in such a branch. Although by Rule 2 vertex $w$ will need to be removed from $G$, we make an exception by temporarily keeping $w$ after we coloured it. The reason is that the presence of $w$ will be helpful for analysing the structure of $(G, L)$ after Rules 2–2 have been applied exhaustively (where we apply Rule 2 only after applying Rules 2–2 exhaustively). In order to do this, we first show the following three claims (proofs omitted).

**Claim 18.** Vertex $w$ is not adjacent to any vertex in $A_2 \cup X_{1,2} \cup X_{1,3}$.

**Claim 19.** The graph $G[X_{1,3} \cup (N(X_{1,3}) \cap A_2) \cup N_3]$ is the disjoint union of one or more complete graphs, each of which consists of either one vertex of $X_{1,3}$ and at most two vertices of $A_2$, or one vertex of $N_3$.

**Claim 20.** For every pair of adjacent vertices $s, t$ with $s \in A_2$ and $t \in N_2$, either $t$ is adjacent to $w$, or $N(s) \cap X_{1,3} \subseteq N(t)$.

We now continue as follows. Recall that $X_{1,3} \neq \emptyset$. Hence there exists a vertex $s \in A_2$ that has a neighbour $r \in X_{1,3}$. As $s \in A_2$, we have that $|L(s)| = 3$. Then, by Rule 2, we find that $s$ has at least two neighbours $t$ and $t'$ not equal to $r$. By Claim 19, we find that neither $t$ nor $t'$ belongs to $X_{1,3} \cup N_3$. We are going to fix an induced 3-vertex path $P^*$ of $G$, over which we will branch, in the following way.

If $t$ and $t'$ are not adjacent, then we let $P^*$ be the induced path in $G$ with vertices $t, s, t'$ in that order. Suppose that $t$ and $t'$ are adjacent. As $G$ is $K_4$-free and $s$ is adjacent to $r, t, t'$, at least one of $t, t'$ is not adjacent to $r$. We may assume without loss of generality that $t$ is not adjacent to $r$.

First assume that $t \in N_2$. Recall that $s$ has a neighbour in $X_{1,3}$, namely $r$, and that $r$ is not adjacent to $t$. We then find that $t$ must be adjacent to $w$ by Claim 20. As $s \in A_2$, we find that $s$ is not adjacent to $w$ by Claim 18. In this case we let $P^*$ be the induced path in $G$ with vertices $s, t, w$ in that order.

Now assume that $t \notin N_2$. Recall that $t \notin N_3$. Hence, $t$ must be in $N_1$. Then, as $t \notin X_{1,3}$ but $t$ is adjacent to a vertex in $A_2$, namely $s$, we find that $t \in X_{1,2}$. Recall that $t' \notin X_{1,3}$. If $t' \in N_1$ then the fact that $t' \notin X_{1,3}$, combined with the fact that $t'$ is adjacent to $s \in A_2$, implies that $t' \in X_{1,2}$. However, by Rule 2 applied on $s, t, t'$, vertex $s$ would have a list of size 1 instead of size 3, a contradiction. Hence, $t' \notin N_1$. As $t' \notin N_3$, this means that $t' \in N_2$. If $t'$ is adjacent to $r$, then $t \in X_{1,2}$ with $L(t) = \{1, 2\}$ and $r \in X_{1,3}$ with $L(r) = \{1, 3\}$ would have the same lists by Rule 2 applied on $r, s, t, t'$, a contradiction. Hence $t'$ is not adjacent to $r$. Then, by Claim 20, we find that $t'$ must be adjacent to $w$. Note that $s$ is not adjacent to $w$ due to Claim 18. In this case we let $P^*$ be the induced path in $G$ with vertices $s, t', w$.
in that order. We conclude that either $P^s = tst'$ or $P^s = stw$ or $P^s = st'w$. We are now ready to apply two more rounds of branching.

### Branching IV. ($O(n)$ branches)
We branch by considering the possibility of removing colour 2 from the list of each vertex in $N(X_{1,3}) \cap A_2$ and all possibilities of choosing a vertex in $N(X_{1,3}) \cap A_2$ and giving it colour 2. In the branch where we removed colour 2 from the list of every vertex in $N(X_{1,3}) \cap A_2$, we obtain that $X_{1,3} = \emptyset$. Hence for that branch we can enter Phase 4. Now consider a branch where we gave some vertex $s \in N(X_{1,3}) \cap A_2$ colour 2. Let $P^s = tst'$ or $P^s = stw$ or $P^s = st'w$. We do some further branching by considering all possibilities of colouring the vertices of $P^s$ that are not equal to the already coloured vertices $s$ and $w$ (should $w$ be a vertex of $P^s$) and all possibilities of giving a colour to the vertex from $N(s) \cap X_{1,3}$ (recall that by Claim 19, $|N(s) \cap X_{1,3}| = 1$). This leads to a total of $O(n)$ branches. We claim that in both branches, $|X_{1,3}|$ has reduced to at most 1 (proof omitted).

### Branching V. ($O(1)$ branches)
We branch by considering both possibilities of colouring the unique vertex of $X_{1,3}$. This leads to two new but smaller instances of List 3-Colouring, in each of which the set $X_{1,3} = \emptyset$. Hence, our algorithm can enter Phase 4.

### Phase 4. Reduce to a set of instances of 2-List Colouring
Recall that in this stage of our algorithm we have an instance $(G, L)$ in which every vertex of $A_1$ has the same list, say $\{1, 2\}$. As $G$ is $(P_2 + P_3)$-free, $G[N_2 \cup N_3]$ is an independent set; otherwise two adjacent vertices of $N_2 \cup N_3$ form, together with $v_1, \ldots, v_5$, an induced $P_2 + P_5$. Hence, we can safely colour each vertex in $A_2$ with colour 3, and afterwards we may apply Theorem 6.

The correctness of our algorithm follows from the description. The branching in the five stages (Branching I-V), yields a total number of $O(n^4\ell)$ branches and each branch we created takes polynomial time to process. Hence, the running time of our algorithm is polynomial.

**Remark.** Except for Phase 4 of our algorithm, all arguments in our proof hold for $(P_3 + P_5)$-free graphs. The difficulty in Phase 4 is that in contrary to the previous phases we cannot use the vertices from $N_0$ to find an induced $P_3 + P_5$ and therefore obtain the contradiction.

### 3 The Hardness Result
We show that 5-Colouring is NP-complete for $(P_3 + P_5)$-free graphs by reducing from the NP-complete problem [32] **Not-All-Equal 3-Satisfiability** with positive literals only, defined as follows: given a set $X = \{x_1, x_2, \ldots, x_n\}$ of logical variables and a set $C = \{C_1, C_2, \ldots, C_m\}$ of 3-literal clauses over $X$ in which all literals are positive, is there a truth assignment for $X$ such that each clause contains at least one true literal and at least one false literal? We call such a truth assignment satisfying.

**Theorem 4 (reiterated).** 5-Colouring is NP-complete for $(P_3 + P_5)$-free graphs.

**Proof.** Proof Sketch. From a given instance $(C, X)$ of **Not-All-Equal 3-Satisfiability** with positive literals only, we first construct a graph $G$ with a list assignment $L$. For each $x_i \in X$ we introduce two vertices $x_i$ and $\overline{x_i}$, which we make adjacent to each other. We say that $x_i$ and $\overline{x_i}$ are of $x$-type. We set $L(x_i) = L(\overline{x_i}) = \{4, 5\}$. For each $C_j \in C$ we introduce
a vertex $C_j$ and a vertex $C'_j$ called the copy of $C_j$. We say that $C_j$ and $C'_j$ are of $C$-type. We set $L(C_j) = L(C'_j) = \{1, 2, 3\}$. We add an edge between each $x$-type vertex and each $C$-type vertex. For each $C_j \in C$ we do as follows. We fix an arbitrary order of the literals in $C_j$. Say $C_j = \{x_g, x_h, x_i\}$ in that order. Then we add six vertices $a_{g,j}$, $a_{h,j}$, $a_{i,j}$, $a'_{g,j}$, $a'_{h,j}$, $a'_{i,j}$ and edges $x_g a_{g,j}$, $a_{g,j} C_j$, $x_h a_{h,j}$, $a_{h,j} C_j$, $x_i a_{i,j}$, $a_{i,j} C_j$ and also edges $\tau_g a'_{g,j}$, $a'_{g,j} C'_j$, $\tau_h a'_{h,j}$, $a'_{h,j} C'_j$, $\tau_i a'_{i,j}$, $a'_{i,j} C'_j$. We say that $a_{g,j}$, $a_{h,j}$, $a_{i,j}$, $a'_{g,j}$, $a'_{h,j}$, $a'_{i,j}$ are of $a$-type. We set $L(a_{g,j}) = L(a'_{g,j}) = \{1, 4\}$, $L(a_{h,j}) = L(a'_{h,j}) = \{2, 4\}$ and $L(a_{i,j}) = L(a'_{i,j}) = \{3, 4\}$.

We now extend $G$ into a graph $G'$ by adding a clique consisting of five new vertices $k_1, \ldots, k_5$, which we say are of $k$-type, and by adding an edge between a vertex $k_l$ and a vertex $u \in V(G)$ if and only if $\ell \notin L(u)$. We can show that $(C, X)$ has a satisfying truth assignment if and only if $G'$ has a 5-colouring, and moreover that $G'$ is $(P_3 + P_5)$-free (proof omitted). As 5-COLOURING belongs to NP, this proves the theorem. ▶

4 Conclusions

By solving two new cases we completed the complexity classifications of 3-COLOURING and LIST 3-COLOURING on $H$-free graphs for graphs $H$ up to seven vertices. We showed that both problems become polynomial-time solvable if $H$ is a linear forest, while they stay NP-complete in all other cases. Recall that $k$-COLOURING ($k \geq 3$) is NP-complete on $H$-free graphs whenever $H$ is not a linear forest. For the case where $H$ is a linear forest, our new NP-hardness result for 5-COLOURING for $(P_3 + P_5)$-free graphs bounds, together with the known NP-hardness results of [20] for 4-COLOURING for $P_4$-free graphs and 5-COLOURING for $P_5$-free graphs, the number of open cases of $k$-COLOURING from above.

For future research we aim to extend our results. In fact we still do not know if there exists a linear forest $H$ such that 3-COLOURING is NP-complete for $H$-free graphs. This is, however, a notorious open problem studied in many papers; for a recent discussion see [16]. It is also open for LIST 3-COLOURING, where an affirmative answer to one of the two problems yields an affirmative answer to the other one [15]. For $k \geq 4$, we emphasize that all open cases involve linear forests $H$ whose connected components are small. For instance, if $H$ has at most six vertices, then the polynomial-time algorithm for 4-PRECOLOURING EXTENSION on $P_6$-free graphs [7, 8] implies that there are only three graphs $H$ with $|V(H)| \leq 6$ for which we do not know the complexity of 4 COLOURING on $H$-free graphs, namely $H \in \{P_1 + P_2 + P_3, P_2 + P_4, 2P_3\}$ (see [14]).

The main difficulty to extend the known complexity results is that hereditary graph classes characterized by a forbidden induced linear forest are still not sufficiently well understood due to their rich structure. We need a better understanding of these graph classes to make further progress on a wide range of problems. For example, INDEPENDENT SET is polynomial-time solvable for $P_6$-free graphs [17], but it is not known if there exists a linear forest $H$ such that it is NP-complete for $H$-free graphs. A similar situation holds for ODD CYCLE TRANSVERSAL and FEEDBACK VERTEX SET and many other problems; see [1] for a survey.

References

2 Flavia Bonomo, Maria Chudnovsky, Peter Maceli, Oliver Schaudt, Maya Stein, and Mingxian Zhong. Three-coloring and list three-coloring of graphs without induced paths on seven vertices. Combinatorica, (in press).
5:12 \textbf{Colouring $(P_r + P_s)$-Free Graphs}


9 Maria Chudnovsky and Juraj Stacho. 3-colorable subclasses of $P_8$-free graphs. \textit{Man.}, 2017.


The Use of a Pruned Modular Decomposition for Maximum Matching Algorithms on Some Graph Classes

Guillaume Ducoffe
ICI – National Institute for Research and Development in Informatics, Bucharest, Romania
The Research Institute of the University of Bucharest ICUB, Bucharest, Romania
guillaume.ducoffe@ici.ro

Alexandru Popa
University of Bucharest, Bucharest, Romania
ICI – National Institute for Research and Development in Informatics, Bucharest, Romania
alexandru.popa@fmi.unibuc.ro

Abstract
We address the following general question: given a graph class $C$ on which we can solve MAXIMUM MATCHING in (quasi) linear time, does the same hold true for the class of graphs that can be modularly decomposed into $C$? As a way to answer this question for distance-hereditary graphs and some other superclasses of cographs, we study the combined effect of modular decomposition with a pruning process over the quotient subgraphs. We remove sequentially from all such subgraphs their so-called one-vertex extensions (i.e., pendant, anti-pendant, twin, universal and isolated vertices). Doing so, we obtain a “pruned modular decomposition”, that can be computed in quasi linear time. Our main result is that if all the pruned quotient subgraphs have bounded order then a maximum matching can be computed in linear time. The latter result strictly extends a recent framework in (Coudert et al., SODA’18). Our work is the first to explain why the existence of some nice ordering over the modules of a graph, instead of just over its vertices, can help to speed up the computation of maximum matchings on some graph classes.

2012 ACM Subject Classification Mathematics of computing → Graph theory, Theory of computation → Design and analysis of algorithms

Keywords and phrases maximum matching, FPT in P, modular decomposition, pruned graphs, one-vertex extensions, $P_4$-structure

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.6


Funding This work was supported by the Institutional research programme PN 1819 “Advanced IT resources to support digital transformation processes in the economy and society - RESINFO-TD” (2018), project PN 1819-01-01 “Modeling, simulation, optimization of complex systems and decision support in new areas of IT&C research”, funded by the Ministry of Research and Innovation, Romania.

1 Introduction

Can we compute a maximum matching in a graph in linear-time? – i.e., computing a maximum set of pairwise disjoint edges in a graph. – Despite considerable years of research and the design of elegant combinatorial and linear programming techniques, the best-known
Pruned modular decomposition and Maximum Matching

algorithms for this fundamental problem have stayed blocked to an $O(m\sqrt{n})$-time complexity on $n$-vertex $m$-edge graphs [22]. Nevertheless, we can use some well-structured graph classes in order to overcome this superlinear barrier for particular cases of graphs. Our work combines two successful approaches for this problem, namely, the use of a vertex-ordering characterization for certain graph classes [5, 10, 21], and a recent technique based on the decomposition of a graph by its modules [9]. We detail these two approaches in what follows, before summarizing our contributions.

1.1 Related work

A cornerstone of most Maximum Matching algorithms is the notion of augmenting paths [2, 15]. However, although we can compute a set of augmenting paths in linear-time [16], this is a tedious task that involves the technical notion of blossoms and this may need to be repeated $\Omega(\sqrt{n})$ times before a maximum matching can be computed [19]. A well-known greedy approach consists in, given some total ordering $(v_1, v_2, \ldots, v_n)$ over the vertices in the graph, to consider the exposed vertices $v_i$ by increasing order, then to try to match them with some exposed neighbour $v_j$ that appears later in the ordering [12]. The vertex $v_j$ can be chosen either arbitrarily or according to some specific rules depending on the graph class we consider. Our initial goal was to extend similar reduction rules to module-orderings.

Modular decomposition. A module in a graph $G = (V, E)$ is any vertex-subset $X$ such that every vertex of $V \setminus X$ is either adjacent to every of $X$ or nonadjacent to every of $X$. The modular decomposition of $G$ is a recursive decomposition of $G$ according to its modules [18]. We postpone its formal definition until Section 2. For now, we only want to stress that the vertices in the “quotient subgraphs” that are outputted by this decomposition represent modules of $G$ (e.g., see Fig. 1 for an insightful illustration). Our main motivation for considering modular decomposition in this note is its recent use in the field of parameterized complexity for polynomial problems. More precisely, let us call modular-width of a graph $G$ the minimum $k \geq 2$ such that every quotient subgraph in the modular decomposition of $G$ is either “degenerate” (i.e., complete or edgeless) or of order at most $k$. With Coudert, we proved in [9] that many “hard” graph problems in P – for which no linear-time algorithm is likely to exist – can be solved in $k^{O(1)}(n + m)$-time on graphs with modular-width at most $k$. In particular, we proposed an $O(k^4 n + m)$-time algorithm for Maximum Matching.

One appealing aspect of our approach in [9] was that, for most problems studied, we obtained a linear-time reduction from the input graph $G$ to some (smaller) quotient subgraph $G'$ in its modular decomposition. – We say that the problem is preserved by quotient. – This paved the way to the design of efficient algorithms for these problems on graph classes with unbounded modular-width, assuming their quotient subgraphs are simple enough w.r.t. the problem at hands. We illustrated this possibility through the case of $(q, q - 3)$-graphs (i.e., graphs where no set of at most $q$ vertices, $q \geq 7$, can induce more than $q - 3$ paths of length four). However, this approach completely fell down for Maximum Matching. Indeed, our Maximum Matching algorithm in [9] works on supergraphs of the quotient graphs that need to be repeatedly updated every time a new augmenting path is computed. Such approach did not help much in exploiting the structure of quotient graphs. We managed to do so for $(q, q - 3)$-graphs only through the help of a deeper structural theorem on the nontrivial modules in this class of graphs. Nevertheless, to take a shameful example, it was not even known before this work whether Maximum Matching could be solved faster than with the state-of-the art algorithms on graphs that can be modularly decomposed into paths!
1.2 Our contributions

We propose pruning rules on the modules in a graph (some of them new and some others revisited) that can be used in order to compute Maximum Matching in linear-time on several new graph classes. More precisely, given a module $M$ in a graph $G = (V, E)$, recall that $M$ is corresponding to some vertex $v_M$ in a quotient graph $G'$ of the modular decomposition of $G$. Assuming $v_M$ is a so-called one-vertex extension in $G'$ (i.e., it is pendant, anti-pendant, universal, isolated or it has a twin), we show that a maximum matching for $G$ can be computed from a maximum matching of $G[M]$ and a maximum matching of $G \setminus M$ efficiently (see Section 4). Our rules are purely structural, in the sense that they only rely on the structural properties of $v_M$ in $G'$ and not on any additional assumption on the nontrivial modules. Some of these rules (e.g., for isolated or universal modules) were first introduced in [9] – although with slightly different correctness proofs. Our main technical contributions in this work are the pruning rules for, respectively, pendant and anti-pendant modules (see Sections 4.2 and 4.3). The latter two cases are surprisingly the most intricate. In particular, they require amongst other techniques: the computation of specified augmenting paths of length up to 7, the addition of some “virtual edges” in other modules, and a careful swapping between some matched and unmatched edges.

Then, we are left with pruning every quotient subgraph in the modular decomposition by sequentially removing the one-vertex extensions. We prove that the resulting “pruned quotient subgraphs” are unique (independent from the removal orderings) and that they can be computed in quasi linear-time using a trie data-structure (Section 3). Furthermore, as a case-study we prove that several superclasses of cographs are totally decomposable w.r.t. this new “pruned modular decomposition”. These classes are further discussed in Section 5. Note that for some of them, such as distance-hereditary graphs, we so obtain the first known linear-time algorithm for Maximum Matching – thereby extending previous partial results obtained for bipartite and chordal distance-hereditary graphs [10]. Our approach actually has similarities with a general greedy scheme applied to distance-hereditary graphs [7]. With slightly more work, we can extend our approach to every graph that can be modularly decomposed into cycles. The case of graphs of bounded modular treewidth [23] is left as an interesting open question.

Definitions and our first results are presented in Section 2. We introduce the pruned modular decomposition in Section 3, where we show that it can be computed in quasi linear-time. Then, the core of the paper is Section 4 where the pruning rules are presented along with their correctness proofs. In particular, we state our main result in Section 4.4. Applications of our approach to some graph classes are discussed in Section 5. Finally, we conclude in Section 6 with some open questions. Due to lack of space, several proofs are omitted. Full proofs can be found in our technical report [14].

2 Preliminaries

For the standard graph terminology, see [3]. We only consider graphs that are finite, simple and unweighted. For any graph $G = (V, E)$ let $n = |V|$ and $m = |E|$. Given a vertex $v \in V$, we denote its (open) neighbourhood by $N_G(v) = \{u \in V \mid \{u, v\} \in E\}$ and its closed neighbourhood by $N_G[v] = N_G(v) \cup \{v\}$. Similarly, we define the neighbourhood of any vertex-subset $S \subseteq V$ as $N_G(S) = \left( \bigcup_{v \in S} N_G(v) \right) \setminus S$. In what follows, we introduce our main algorithmic tool for the paper as well as the graph problems we study.
Pruned modular decomposition and Maximum Matching

Modular decomposition

A module in a graph \( G = (V,E) \) is any subset \( M \subseteq V(G) \) such that for any \( u,v \in M \) we have \( N_G(v) \setminus M = N_G(u) \setminus M \). There are trivial examples of modules such as \( \emptyset, V, \) and \( \{v\} \) for every \( v \in V \). Let \( \mathcal{P} = \{M_1, M_2, \ldots, M_p\} \) be a partition of the vertex-set \( V \). If for every \( 1 \leq i \leq p \), \( M_i \) is a module of \( G \), then we call \( \mathcal{P} \) a modular partition of \( G \). By abuse of notation, we will sometimes identify a module \( M_i \) with the induced subgraph \( H_i = G[M_i] \), i.e., we will write \( \mathcal{P} = \{H_1, H_2, \ldots, H_p\} \). The quotient subgraph \( G/\mathcal{P} \) has vertex-set \( \mathcal{P} \), and there is an edge between every two modules \( M_i, M_j \in \mathcal{P} \) such that \( M_i \times M_j \subseteq E \). Conversely, let \( G' = (V', E') \) be a graph and let \( \mathcal{P} = \{H_1, H_2, \ldots, H_p\} \) be a collection of subgraphs. The substitution graph \( G'(\mathcal{P}) \) is obtained from \( G' \) by replacing every vertex \( v_i \in V' \) with a module inducing \( H_i \). In particular, for \( G' = \text{def} \ G/\mathcal{P} \) we have that \( G'(\mathcal{P}) = G \).

We say that \( G \) is prime if its only modules are trivial (i.e., \( \emptyset, V, \) and the singletons \( \{v\} \)). We call a module \( M \) strong if it does not overlap any other module, i.e., for any module \( M' \) of \( G \), either one of \( M \) or \( M' \) is contained in the other or \( M \) and \( M' \) do not intersect. Let \( \mathcal{M}(G) \) be the family of all inclusion wise maximal strong modules of \( G \) that are proper subsets of \( V \). The family \( \mathcal{M}(G) \) is a modular partition of \( G \) [18], and so, we can define \( G' = G/\mathcal{M}(G) \). The following structure theorem is due to Gallai.

**Theorem 1** ([17]). For an arbitrary graph \( G \) exactly one of the following conditions is satisfied.
1. \( G \) is disconnected;
2. its complement \( \overline{G} \) is disconnected;
3. or its quotient graph \( G' = G/\mathcal{M}(G) \) is prime for modular decomposition.

We now formally define the modular decomposition of \( G \) – introduced earlier in Section 1. We output the quotient graph \( G' = G/\mathcal{M}(G) \) and, for any strong module \( M \in \mathcal{M}(G) \) that is nontrivial (possibly none if \( G = G' \)), we also output the modular decomposition of \( G[M] \). By Theorem 1 the subgraphs from the modular decomposition are either edgeless, complete, or prime for modular decomposition. See Fig. 1 for an example. The modular decomposition of a given graph \( G = (V,E) \) can be computed in linear-time [25]. There are many graph classes that can be characterized using the modular decomposition. In particular, \( G \) is a cograph if and only if every quotient subgraph in its modular decomposition is either complete or disconnected [8].

Maximum Matching

A matching in a graph is defined as a set of edges with pairwise disjoint end vertices. The maximum cardinality of a matching in a given graph \( G = (V,E) \) is denoted by \( \mu(G) \).
We remind the reader that Maximum Matching can be solved in \( O(m\sqrt{n}) \)-time on general graphs [22] – although we do not use this result directly in our paper. Furthermore, let \( G = (V, E) \) be a graph and let \( F \subseteq E \) be a matching of \( G \). We call a vertex matched if it is incident to an edge of \( F \), and exposed otherwise. Then, we define an \( F \)-augmenting path as a path where the two ends are exposed, and the edges belong alternatively to \( F \) and not to \( F \). It is well-known and easy to check that, given an \( F \)-augmenting path \( P \), the matching \( E(P)\Delta F \) (obtained by symmetric difference on the edges) has larger cardinality than \( F \).

**Problem 2 (Maximum Matching).**

**Input:** A graph \( G = (V, E) \).

**Output:** A matching of \( G \) with maximum cardinality.

In this paper, we will consider an intermediate matching problem, first introduced in [9].

**Problem 4 (Module Matching).**

**Input:** A graph \( G' = (V', E') \) with the following additional information:
- a collection of subgraphs \( \mathcal{P} = \{H_1, H_2, \ldots, H_p\} \);
- a collection \( \mathcal{F} = \{F_1, F_2, \ldots, F_p\} \),
  with \( F_i \) being a maximum matching of \( H_i \) for every \( i \).

**Output:** A matching of \( G = G'(\mathcal{P}) \) with maximum cardinality.

A natural choice for Module Matching would be to take \( \mathcal{P} = \mathcal{M}(G) \). However, we will allow \( \mathcal{P} \) to take different values for our reduction rules.

**Additional notations.** Let \( \langle G', \mathcal{P}, \mathcal{F} \rangle \) be any instance of Module Matching. The order of \( G' \), equivalently the cardinality of \( \mathcal{P} \), is denoted by \( p \). For every \( 1 \leq i \leq p \) let \( M_i = V(H_i) \) and let \( n_i = |M_i| \) be the order of \( H_i \). We denote \( \delta_i = |E(M_i, \overline{M_i})| \) the size of the cut \( E(M_i, \overline{M_i}) \) with all the edges between \( M_i \) and \( N_G(M_i) \). In particular, we have \( \delta_i = \sum_{v \in N_G(M_i)} h_i n_j \). Let us define \( \Delta n(G') = \sum_{i=1}^{p} \delta_i \). We will omit the dependency in \( G' \) if it is clear from the context. Finally, let \( \Delta \mu = \mu(G) - \sum_{i=1}^{p} \mu(H_i) \).

Our framework is based on the following lemma (inspired from [9]).

**Lemma 5.** Let \( G = (V, E) \) be a graph. Suppose that for every \( H' \) in the modular decomposition of \( G \) we can solve Module Matching on any instance \( \langle H', \mathcal{P}, \mathcal{F} \rangle \) in time \( T(p, \Delta m, \Delta \mu) \), where \( T \) is a subadditive function\(^1\). Then, we can solve Maximum Matching on \( G \) in time \( O(T(\mathcal{O}(n), m, n)) \).

An important observation for our subsequent analysis is that, given any module \( M \) of a graph \( G \), the internal structure of \( G[M] \) has no more relevance after we computed a maximum matching \( F_M \) for this subgraph. More precisely, we will use the following lemma:

**Lemma 6** [9]. Let \( M \) be a module of \( G = (V, E) \), let \( G[M] = (M, E_M) \) and let \( F_M \subseteq E_M \) be a maximum matching of \( G[M] \). Then, every maximum matching of \( G'[M] = (V, (E \setminus E_M) \cup F_M) \) is a maximum matching of \( G \).

By Lemma 6 we can modify our algorithmic framework as follows. For every instance \( \langle G', \mathcal{P}, \mathcal{F} \rangle \) for Module Matching, we can assume that \( H_i = (M_i, F_i) \) for every \( 1 \leq i \leq p \).

\(^1\) We stress that every polynomial function is subadditive.
**Data structures.** Finally, let \( \langle G', \mathcal{P}, \mathcal{F} \rangle \) be any instance for Module Matching. A canonical ordering of \( H_i \) w.r.t. \( F_i \) is a total ordering over \( V(H_i) \) such that the exposed vertices appear first, and every two vertices that are matched together are consecutive. In what follows, we will assume that we have access to a canonical ordering for every \( i \). Such orderings can be computed in time \( \mathcal{O}(\sum_i |M_i| + |F_i|) \) by scanning all the modules and the matchings in \( \mathcal{F} \), that is an \( \mathcal{O}(\Delta m) \) provided \( G' \) has no isolated vertex.

Furthermore, let \( F \) be a (not necessarily maximum) matching for the subdivision \( G = G'(\mathcal{P}) \). We will make the standard assumption that, for every \( v \in V(G) \), we can decide in constant-time whether \( v \) is matched by \( F \), and if so, we can also access in constant-time to the vertex matched with \( v \).

### 3 A pruned modular decomposition

In this section, we introduce a pruning process over the quotient subgraphs, that we use in order to refine the modular decomposition.

**Definition 7.** Let \( G = (V, E) \) be a graph. We call \( v \in V \) a one-vertex extension if it falls in one of the following cases:

- \( N_G[v] = V \) (universal) or \( N_G(v) = \emptyset \) (isolated);
- \( N_G[v] = V \setminus u \) (anti-pendant) or \( N_G(v) = \{u\} \) (pendant), for some \( u \in V \setminus v \);
- \( N_G[v] = N_G[u] \) (true twin) or \( N_G(v) = N_G(u) \) (false twin), for some \( u \in V \setminus v \).

A pruned subgraph of \( G \) is obtained from \( G \) by sequentially removing one-vertex extensions (in the current subgraph) until it can no more be done. This terminology was introduced in [20], where they only considered the removals of twin and pendant vertices. Also, the clique-width of graphs that are totally decomposed by the above pruning process (i.e., with their pruned subgraph being a singleton) was studied in [24] \(^2\). Our contribution in this part is twofold. First, we show that the gotten subgraph is “almost” independent of the removal ordering, i.e., there is a unique pruned subgraph of \( G \) (up to isomorphism). The latter can be derived from the following (easy) lemma:

**Lemma 8.** Let \( G = (V, E) \) be a graph and let \( v, v' \in V \) be one-vertex extensions of \( G \). If \( v, v' \) are not pairwise twins then \( v' \) is a one-vertex extension of \( G \setminus v \).

**Corollary 9.** Every graph \( G = (V, E) \) has a unique pruned subgraph up to isomorphism.

For many graph classes a pruning sequence can be computed in linear-time. We observe that the same can be done for any graph (up to a logarithmic factor).

**Proposition 10.** For every graph \( G = (V, E) \), we can compute a pruned subgraph in \( \mathcal{O}(n + m \log n) \)-time.

**Proof.** By Corollary 9, we are left with greedily searching for, then eliminating, the one-vertex extensions. We can compute the ordered degree sequence of \( G \) in \( \mathcal{O}(n + m) \)-time. Furthermore, after any vertex \( v \) is eliminated, we can update this sequence in \( \mathcal{O}(|N(v)|) \)-time. Hence, up to a total update time in \( \mathcal{O}(n + m) \), at any step we can detect and remove in constant-time any vertex that is either universal, isolated, pendant or anti-pendant. Finally, in [20] they proposed a trie data-structure supporting the following two operations: suppression of a vertex; and detection of true or false twins (if any). The total time for all the operations on this data-structure is in \( \mathcal{O}(n + m \log n) \) [20].

---

\(^2\) Anti-twins are also defined as one-vertex extensions in [24]. Their integration to this framework remains to be done.
We will term “pruned modular decomposition” of a graph $G$ the collection of the pruned subgraphs for all the quotient subgraphs in the modular decomposition of $G$. Note that there is a unique pruned modular decomposition of $G$ (up to isomorphism) and that it can be computed in $O(n + m \log n)$-time by Proposition 10 (applied to every quotient subgraph in the modular decomposition separately). Furthermore, we remark that most cases of one-vertex extensions imply the existence of non trivial modules, and so, they cannot exist in the prime quotient subgraphs of the modular decomposition. Nevertheless, such vertices may appear after removal of pendant or anti-pendant vertices (e.g., in the bull graph).

4 Reduction rules

Let $(G', P, F)$ be any instance of Module Matching. Suppose that $v_1$, the vertex corresponding to $M_1$ in $G'$, is a one-vertex extension. Under this assumption, we present reduction rules to a smaller instance $(G^*, P^*, F^*)$ where $|P^*| < |P|$. Each rule can be implemented to run in $O(\Delta m(G') - \Delta m(G^*))$-time. Due to lack of space, we skip the complexity analysis.

In Section 4.1 we recall the rules introduced in [9] for universal and isolated modules (explicitly) and for false or true twin modules (implicitly). Our main technical contributions are the reduction rules for pendant and anti-pendant modules (in Sections 4.2 and 4.3, respectively), which are surprisingly the most intricate. Finally, we end this section stating our main result (Theorem 29).

4.1 Simple cases

We introduce two local operations on a matching, first used in [26] for cographs. Let $F \subseteq E$ be a matching and let $M \subseteq V$ be a module.

- **Operation 11 (MATCH).** While there are $x \in M$, $y \in N(M)$ exposed, add $\{x, y\}$ to $F$.
- **Operation 12 (SPLIT).** While there are $x, x' \in M$, $y, y' \in N(M)$ such that $x$ and $x'$ are exposed, and $\{y, y'\} \in F$, replace $\{y, y'\}$ in $F$ by $\{x, y\}$, $\{x', y'\}$.

Let $G = H_1 \oplus H_2$ be the join of the two graphs $H_1, H_2$ and let $F_1, F_2$ be maximum matchings for $H_1, H_2$, respectively. The “MATCH and SPLIT” technique consists in applying Operations 11 then 12 to $M = V(H_1)$ and $F = F_1 \cup F_2$, thereby obtaining a new matching $F'$, then to $M = V(H_2)$ and $F = F'$. Based on this technique, we design the following rules:

- **Reduction rule 13 (see also [9]).** Suppose $v_1$ is isolated in $G'$. We set $G^* = G' \setminus v_1$, $P^* = P \setminus \{H_1\}$, and $F^* = F \setminus \{F_1\}$. Furthermore, let $F^*$ be a maximum matching of $G^*(P^*) = G[V \setminus M_1]$. We output $F^* \cup F_1$.

- **Reduction rule 14 (see also [9]).** Suppose $v_1$ is universal in $G'$. We set $G^* = G \setminus v_1$, $P^* = P \setminus \{H_1\}$, $F^* = F \setminus \{F_1\}$. Furthermore, let $F^*$ be a maximum matching of the subdivision $G^*(P^*) = G[V \setminus M_1]$. We apply the “MATCH and SPLIT” technique to $M_1, F_1$ with $V \setminus M_1, F^*$.

- **Reduction rule 15.** Suppose $v_1, v_2$ are false twins in $G'$. We set $G^* = G \setminus v_1$, $P^* = \{H_1 \cup H_2\} \cup (P \setminus \{H_1, H_2\})$, $F^* = \{F_1 \cup F_2\} \cup (F \setminus \{F_1, F_2\})$. We output a maximum matching of $G^*(P^*) = G$.

- **Reduction rule 16.** Suppose $v_1, v_2$ are true twins in $G'$. Let $F_2^*$ be the matching of $H_1 \oplus H_2$ obtained from the “MATCH and SPLIT” technique applied to $M_1, F_1$ with $M_2, F_2$. We set $G^* = G \setminus v_1$, $P^* = \{H_1 \oplus H_2\} \cup (P \setminus \{H_1, H_2\})$, $F^* = \{F_2^*\} \cup (F \setminus \{F_1, F_2\})$. We output a maximum matching of $G^*(P^*) = G$. 
4.2 Anti-pendant

Suppose \( v_1 \) is anti-pendant in \( G' \). W.l.o.g., \( v_2 \) is the unique vertex that is nonadjacent to \( v_1 \) in \( G' \). By Lemma 6, we can also assume w.l.o.g. that \( E(H_1) = F_i \) for every \( i \). In this situation, we start applying Reduction rule 13, i.e., we set \( G^* = G' \setminus v_1 \), \( F^* = P \setminus \{H_1\} \), \( F^* = F \setminus \{F_1\} \). Then, we obtain a maximum matching \( F^* \) of \( G \setminus \cup \{M_i\} \) (i.e., by applying our reduction rules to this new instance). Finally, from \( F_1 \) and \( F^* \), we compute a maximum matching \( F \) of \( G \), using an intricate procedure. We detail this procedure next.

First phase: pre-processing. Our correctness proofs in what follows will assume that some additional properties hold on the matched vertices in \( F^* \). So, we start correcting the initial matching \( F^* \) so that it is the case. For that, we introduce two “swapping” operations. Recall that \( v_2 \) is the unique vertex that is nonadjacent to \( v_1 \) in \( G' \).

- **Operation 17** (REPAIR). While there exist \( x_2, y_2 \in M_2 \) such that \( \{x_2, y_2\} \in F_2 \) and \( y_2 \) is exposed in \( F^* \), we replace any edge \( \{x_2, w\} \in F^* \) by \( \{x_2, y_2\} \).

- **Operation 18** (ATTRACT). While there exist \( x_2 \in M_2 \) exposed and \( \{u, w\} \in F^* \) such that \( u \in N(G(M_2)), w \notin M_2 \), we replace \( \{u, w\} \) by \( \{u, x_2\} \).

  Let \( F^{(0)} = F_1 \cup F^* \). Summarizing, we get:

  ▶ **Definition 19.** A matching \( F \) of \( G \) is good if it satisfies the following two properties:
  1. every vertex matched by \( F_1 \cup F_2 \) is also matched by \( F \);
  2. either every vertex in \( M_2 \) is matched, or there is no matched edge in \( N(G(M_2)) \times N(G(M_1)) \).

- **Fact 20.** \( F^{(0)} \) is a good matching of \( G \).

Main phase: a modified Match and Split. We now apply the following three operations sequentially:

1. **MATCH(M_1, F^{(0)})** (Operation 11). Doing so, we obtain a larger good matching \( F^{(1)} \).
2. **SPLIT(M_1, F^{(1)})** (Operation 12). Doing so, we obtain a larger good matching \( F^{(2)} \).
3. the operation UNBREAK, defined in what follows (see also Fig. 2 for an illustration):

   ▶ **Operation 21** (UNBREAK). While there exist \( x_1 \in M_1 \) and \( x_2 \in M_1 \cup M_2 \) exposed, and there also exist \( \{y_2, z_2\} \in F_2 \setminus F^{(2)} \) we replace any two edges \( \{y_2, u\}, \{z_2, w\} \in F^{(2)} \) by the three edges \( \{x_2, u\}, \{y_2, z_2\}, \{w, x_1\} \).

   We stress that the two edges \( \{y_2, u\}, \{z_2, w\} \in F^{(2)} \) always exist since \( F^{(2)} \) is a good matching of \( G \). Furthermore doing so, we obtain a larger matching \( F^{(3)} \).

The resulting matching \( F^{(3)} \) is not necessarily maximum. However, this matching satisfies the following crucial property:

- **Lemma 22.** No vertex of \( M_1 \) can be an end in an \( F^{(3)} \)-augmenting path.
Finalization phase: breaking some edges in $F_1$. Intuitively, the matching $F^{(3)}$ may not be maximum because we sometimes need to borrow some edges of $F_1$ in augmenting paths. So, we complete our procedure by performing the following two operations: Let $U_1$ contain all the exposed vertices in $N(M_1)$. Consider the subgraph $G[M_1 \cup U_1] = G[M_1] \oplus G[U_1]$. The set $U_1$ is a module of this subgraph. We apply $\text{Split}(U_1, F^{(3)})$ in $G[M_1 \cup U_1]$. Doing so, we obtain a larger good matching $F^{(4)}$. Then, we apply $\text{LocalAug}$, defined next (see also Fig. 3 for an illustration):

- **Operation 23 (LocalAug).** While there exist $x_2 \in M_2$ and $c \in N(M_1)$ exposed, and there also exist $\{x_1, y_1\} \in F_1 \cap F^{(4)}$ and $\{y_2, z_2\} \in F_2 \setminus F^{(4)}$, we do the following:
  - we remove $\{x_1, y_1\}$ and any edge $\{a, y_2\}, \{b, z_2\}$ from $F^{(4)}$;
  - we add $\{x_2, a\}, \{y_2, z_2\}, \{b, x_1\}$ and $\{y_1, c\}$ in $F^{(4)}$.

We stress that the two edges $\{y_2, a\}, \{z_2, b\} \in F^{(4)}$ always exist since $F^{(4)}$ is a good matching of $G$. Furthermore doing so, we obtain a larger matching $F^{(5)}$.

- **Lemma 24.** $F^{(5)}$ is a maximum-cardinality matching of $G$.

4.3 Pendant

Suppose $v_1$ is pendant in $G'$. W.l.o.g., $v_2$ is the unique vertex that is adjacent to $v_1$ in $G'$. This last case is arguably more complex than the others since it requires both a pre-processing and a post-processing treatment on the matching.

First phase: greedy matching. We apply the “match and split” technique to $M_1$. Doing so, we obtain a set $F_{1,2}$ of matched edges between $M_1$ and $M_2$. We remove $V(F_{1,2})$, the set of vertices incident to an edge of $F_{1,2}$, from $G$. Then, there are three cases. If $M_2 \subseteq V(F_{1,2})$ then $M_1 \setminus V(F_{1,2})$ is isolated. We apply Reduction rule 13. If $M_1 \subseteq V(F_{1,2})$ then $M_1$ is already eliminated. The interesting case is when both $M_1 \setminus V(F_{1,2})$ and $M_2 \setminus V(F_{1,2})$ are nonempty. In particular, suppose there remains an exposed vertex $x_1 \in M_1 \setminus V(F_{1,2})$. Since $M_2 \setminus V(F_{1,2}) \neq \emptyset$, there exists $\{x_2, y_2\} \in F_2$ such that $x_2, y_2 \not\in V(F_{1,2})$. We remove $x_1$ from $M_1$, $x_2$ from $M_2$, $\{x_2, y_2\}$ from $F_2$ and then we add $\{x_1, x_2\}$ in $F_{1,2}$. Our first result in this section is that there always exists an optimal solution that contains $F_{1,2}$. This justifies a posteriori the removal of $V(F_{1,2})$ from $G$.

- **Lemma 25.** There is a maximum matching of $G$ that contains all edges in $F_{1,2}$.

We stress that during this phase, all the operations except maybe the last one increase the cardinality of the matching. Furthermore, the only possible operation that does not increase the cardinality of the matching is the replacement of an edge in $F_2$ by an edge in $F_{1,2}$. Doing so, either we fall in one of the two pathological cases $M_1 \subseteq V(F_{1,2})$ or $M_2 \subseteq V(F_{1,2})$ (easy to solve), or then we obtain through the replacement operation the following stronger property:

- **Property 26.** All vertices in $M_1$ are matched by $F_1$. 

![Figure 3](image-url)
We will assume Property 26 to be true for the remaining of this section.

**Second phase: virtual split edges.** We complete the previous phase by performing a Split between $M_2, M_1$ (Operation 12). That is, while there exist two exposed vertices $x_2, y_2 \in M_2$ and a matched edge $\{x_1, y_1\} \in F_1$ we replace $\{x_1, y_1\}$ by $\{x_1, x_2\}, \{y_1, y_2\}$ in the current matching. However, we encode the Split operation using virtual edges in $H_2$. Formally, we add a virtual edge $\{x_2, y_2\}$ in $H_2$ that is labeled by the corresponding edge $\{x_1, y_1\} \in F_1$. Let $H_2^\ast$ and $F_2^\ast$ be obtained from $H_2$ and $F_2$ by adding all the virtual edges. We set $G^\ast = G' \setminus v_1$, $\mathcal{P}^\ast = \{H_2^\ast\} \cup (\mathcal{P} \setminus \{H_1, H_2\})$ and $F^\ast = \{F_2^\ast\} \cup (F \setminus \{F_1, F_2\})$.

Intuitively, virtual edges are used in order to shorten the augmenting paths crossing $M_1$.

**Third phase: post-processing.** Let $F^\ast$ be a maximum-cardinality matching of the subdivision $G^\ast(\mathcal{P}^\ast)$ (i.e., obtained by applying our reduction rules to the new instance). We construct a matching $F$ for $G$ as follows. We add in $F$ all the non virtual edges in $F^\ast$. For every virtual edge $\{x_2, y_2\}$, let $\{x_1, y_1\} \in F_1$ be its label. If $\{x_2, y_2\} \in F^\ast$ then we add $\{x_1, y_2\}, \{x_2, y_1\}$ in $F$, otherwise we add $\{x_1, y_1\}$ in $F$. In the first case, we say that we confirm the Split operation, whereas in the second case we say that we cancel it. Finally, we complete $F$ with all the edges of $F_1$ that do not label any virtual edge (i.e., unused during the second phase).

▶ **Lemma 27.** $F$ is a maximum-cardinality matching of $G$.

The above result is proved by contrapositive. More precisely, we prove intricate properties on the intersection of shortest augmenting paths with pendant modules. Using these properties and the virtual edges, we could transform any shortest $F$-augmenting path into an $F^\ast$-augmenting path, a contradiction.

### 4.4 Main result

Our framework consists in applying any reduction rule presented in this section until it can no more be done. Then, we rely on the following result:

▶ **Theorem 28 ([9]).** We can solve Module Matching for $(G', \mathcal{P}, F)$ in $O(\Delta \mu \cdot p^4)$-time.

We are now ready to state our main result in this paper (the proof of which directly follows from all the previous results in this section).

▶ **Theorem 29.** Let $G = (V, E)$ be a graph. Suppose that, for every prime subgraph $H'$ in the modular decomposition of $G$, its pruned subgraph has order at most $k$. Then, we can solve Maximum Matching for $G$ in $O(k^4 \cdot n + m \log n)$-time.

### 5 Applications

We conclude this paper presenting applications and refinements of our main result to some graph classes. Recall that cographs are exactly the graphs that are totally decomposable by modular decomposition [8]. We start showing that several distinct generalizations of cographs in the literature are totally decomposable by the pruned modular decomposition.
Distance-hereditary graphs. A graph $G = (V, E)$ is distance-hereditary if it can be reduced to a singleton by pruning sequentially the pendant vertices and twin vertices [1]. Conversely, $G$ is co-distance hereditary if it is the complement of a distance-hereditary graph, i.e., it can be reduced to a singleton by pruning sequentially the anti-pan vertices and twin vertices. In both cases, the corresponding pruning sequence can be computed in linear-time [11, 13]. Therefore, we can derive the following result from our framework:

**Proposition 30.** We can solve Maximum Matching in linear-time on graphs that can be modularly decomposed into distance-hereditary graphs and co-distance hereditary graphs.

Trees are a special subclass of distance-hereditary graphs. We say that a graph has modular treewidth at most $k$ if every prime quotient subgraph in its modular decomposition has treewidth at most $k$. In particular, graphs with modular treewidth at most one are exactly the graphs that can be modularly decomposed into trees\(^3\). We stress the following consequence of Proposition 30:

**Corollary 31.** We can solve Maximum Matching in linear-time on graphs with modular-treewidth at most one.

The case of graphs with modular treewidth $k \geq 2$ is left as an intriguing open question.

Tree-perfect graphs. Two graphs $G_1, G_2$ are $P_4$-isomorphic if there exists a bijection from $G_1$ to $G_2$ such that a 4-tuple induces a $P_4$ in $G_1$ if and only if its image in $G_2$ also induces a $P_4$ [6]. The notion of $P_4$-isomorphism plays an important role in the study of perfect graphs. A graph is tree-perfect if it is $P_4$-isomorphic to a tree [4]. We prove the following result:

**Proposition 32.** Tree-perfect graphs are totally decomposable by the pruned modular decomposition. In particular, we can solve Maximum Matching in linear-time on tree-perfect graphs.

Our proof is based on a deep structural characterization of tree-perfect graphs [4].

The case of unicycles. We end up this section with a refinement of our framework for the special case of unicyclic quotient graphs (a.k.a., graphs with exactly one cycle).

**Proposition 33.** We can solve Maximum Matching in linear-time on the graphs that can be modularly decomposed into unicycles.

For that, we reduce the case of unicycles to the case of cycles (removing pendant modules). Then, we test for all possible numbers of matched edges between two adjacent modules. Doing so, we reduce the case of cycles to the case of paths.

6 Open problems

The pruned modular decomposition happens to be an interesting add up in the study of Maximum Matching algorithms. An exhaustive study of its other algorithmic applications remains to be done. Moreover, another interesting question is to characterize the graphs that are totally decomposable by this new decomposition. We note that our pruning process can

\(^3\) Our definition is more restricted than the one in [23] since they only impose the quotient subgraph $G'$ to have bounded treewidth.
be seen as a repeated update of the modular decomposition of a graph after some specified modules (pendant, anti-pendant) are removed. However, we can only detect a restricted family of these new modules (i.e., universal, isolated, twins). A fully dynamic modular decomposition algorithm could be helpful in order to further refine our framework.

References


A Novel Algorithm for the All-Best-Swap-Edge Problem on Tree Spanners

Davide Bilò
Department of Humanities and Social Sciences, University of Sassari, Italy
davidebil@uniss.it
https://orcid.org/0000-0003-3169-4300

Kleitos Papadopoulos
InSPIRE, Agamemnonos 20, Nicosia, 1041, Cyprus
kleitospa@gmail.com
https://orcid.org/0000-0002-7086-0335

Abstract
Given a 2-edge connected, unweighted, and undirected graph $G$ with $n$ vertices and $m$ edges, a $\sigma$-tree spanner is a spanning tree $T$ of $G$ in which the ratio between the distance in $T$ of any pair of vertices and the corresponding distance in $G$ is upper bounded by $\sigma$. The minimum value of $\sigma$ for which $T$ is a $\sigma$-tree spanner of $G$ is also called the stretch factor of $T$. We address the fault-tolerant scenario in which each edge $e$ of a given tree spanner may temporarily fail and has to be replaced by a best swap edge, i.e., an edge that reconnects $T - e$ at a minimum stretch factor. More precisely, we design an $O(n^2)$ time and space algorithm that computes a best swap edge of every tree edge. Previously, an $O(n^2 \log n)$ time and $O(n^2 + m \log n)$ space algorithm was known for edge-weighted graphs [Bilò et al., ISAAC 2017]. Even if our improvements on both the time and space complexities are of a polylogarithmic factor, we stress the fact that the design of a $o(n^2)$ time and space algorithm would be considered a breakthrough.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases Transient edge failure, best swap edges, tree spanner

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.7

1 Introduction

Given a 2-edge connected, unweighted, and undirected graph $G$ with $n$ vertices and $m$ edges, a $\sigma$-tree spanner is a spanning tree $T$ of $G$ in which the ratio between the distance in $T$ of any pair of vertices and the corresponding distance in $G$ is upper bounded by $\sigma$. The minimum value of $\sigma$ for which $T$ is a $\sigma$-tree spanner of $G$ is also called the stretch factor of $T$. The stretch factor of a tree spanner is a measure of how the all-to-all distances degrade w.r.t. the underlying communication graph if we want to sparsify it. Therefore, tree spanners find several applications in the network design problem area as well as in the area of distributed algorithms (see also [13, 16] for some additional practical motivations).

Unfortunately, tree-based network infrastructures are highly sensitive to even a single transient link failure, since this always results in a network disconnection. Furthermore, when these events occur, the computational costs for rearranging the network flow of information from scratch (i.e., recomputing a new tree spanner with small stretch factor, reconfiguring the routing tables, etc.) can be extremely high. Therefore, in such cases it is enough to promptly reestablish the network connectivity by the addition of a swap edge, i.e., a link that temporarily substitutes the failed edge.

© Davide Bilò and Kleitos Papadopoulos; licensed under Creative Commons License CC-BY 29th International Symposium on Algorithms and Computation (ISAAC 2018).
Editors: Wen-Lian Hsu, Der-Tsai Lee, and Chung-Shou Liao; Article No. 7; pp. 7:1–7:12
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
A Novel Algorithm for the ABSE Problem on Tree Spanners

Table 1 summarizes the state of the art for the ABSE problem on tree spanners. The naive algorithm works as follows: for each edge $e$ of the tree spanner $T$ (that are $O(n)$), we look at all the possible swap edges (that are $O(m)$) and, for each swap edge $f$, we compute the stretch factor of $T$ where $e$ is swapped with $f$ (this requires $O(n^2)$). We observe that the naive algorithm needs to store the all-to-all (post-failure) distances in $G - e$.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>weighted graphs</th>
<th>unweighted graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>naive</td>
<td>$\Theta(n^3m)$</td>
<td>$\Theta(n^3m)$</td>
</tr>
<tr>
<td>Das et al. [7]</td>
<td>$O(m^2 \log n)$</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>Bilò et al. [2]</td>
<td>$O(m^2 \log \alpha(m, n))$</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>Bilò et al. [3]</td>
<td>$O(n^2 \log^4 n)$</td>
<td>$O(n^2 + m \log^2 n)$</td>
</tr>
<tr>
<td>this paper</td>
<td>-</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

In this paper we address the fault-tolerant scenario in which each edge $e$ of a given tree spanner may undergo a transient failure and has to be replaced by a best swap edge, i.e. an edge that reconnects $T - e$ at a minimum stretch factor. More precisely, we design an $O(n^2)$ time and space algorithm that computes all the best swap edges (ABSE for short) in unweighted graphs, that is a best swap edge for every edge of $T$. Previously, an $O(n^2 \log^2 n)$ time and $O(n^2 + m \log^2 n)$ space algorithm was known for edge-weighted graphs. Even though the overall improvements in both the time and space complexities are of a polylogarithmic factor, we stress the fact that designing an $O(n^2)$ time and space algorithm would be considered a breakthrough in this field (see [3]). Furthermore, the approach proposed in this paper uses only one technique provided in [2]; all the remaining ideas are totally new and are at the core of the design of both a time and space efficient algorithm. Our algorithm is also easy to implement and makes use of very simple data structures.

1.1 Related work

The ABSE problem on tree spanners has been introduced by Das et al. in [7], where the authors designed two algorithms for both the weighted and the unweighted case, running in $O(m^2 \log n)$ and $O(n^3)$ time, respectively, and using $O(m)$ and $O(n^2)$ space, respectively. Subsequently, Bilò et al. [2] improved both results by providing two efficient linear-space solutions for both the weighted and the unweighted case, running in $O(m^2 \log \alpha(m, n))$ and $O(mn \log n)$ time, respectively. Recently, in [3] the authors designed a very clever recursive algorithm that uses centroid-decomposition techniques and lower envelope data structures to solve the ABSE problem on tree spanners in $O(n^2 \log^4 n)$ time and $O(n^2 + m \log^2 n)$ space. Table 1 summarizes the state of the art for the ABSE problem on tree spanners.

1.2 Other related work on ABSE

The ABSE problems in spanning trees have received a lot of attention from the algorithmic community. The most famous and first studied ABSE problem was on minimum spanning trees, where the quality of a swap edge is measured w.r.t. the overall cost of the resulting tree (i.e., sum of the edge weights). This problem, a.k.a. sensitivity analysis problem on minimum spanning trees, can be solved in $O(m \log \alpha(m, n))$ time [15], where $\alpha$ denotes the inverse of the Ackermann function. In the minimum diameter spanning tree a quality of a swap edge is measured w.r.t. the diameter of the swap tree [12, 14]. Here the ABSE problem can also be solved in $O(m \log \alpha(m, n))$ time [6]. In the minimum routing-cost spanning tree,
the best swap minimizes the overall sum of the all-to-all distances of the swap tree [18]. The fastest algorithm for solving the ABSE problem in this case has a running time of \(O(m^2\log(n))\) [5]. Concerning the single-source shortest-path tree, several criteria for measuring the quality of a swap edge have been considered. The most important ones are:

- the maximum or the average distance from the root; here the corresponding ABSE problems can be solved in \(O(m\log(n))\) time (see [6]) and \(O(m\log(n)\log\log(n))\) time (see [17]), respectively;
- the maximum and the average stretch factor from the root for which the corresponding ABSE problems have been solved in \(O(mn^2\log(n))\) and \(O(mn\log(n))\) time, respectively [4].

Finally, the ABSE problems have also been studied in a distributed setting [8, 9, 10].

2 Preliminary definitions

Let \(G = (V(G), E(G))\) be a 2-edge-connected, unweighted, and undirected graph of \(n\) vertices and \(m\) edges, respectively, and let \(T\) be a spanning tree of \(G\). Given an edge \(e \in E(G)\), we denote by \(G - e = (V(G), E(G) \setminus \{e\})\) the graph obtained after the removal of \(e\) from \(G\). Given an edge \(e \in E(T)\), let \(S(e)\) denote the set of all the swap edges of \(e\), i.e., all edges in \(E(G) \setminus \{e\}\) whose endpoints belong to two different connected components of \(T - e\). For any \(e \in E(T)\) and \(f \in S(e)\), let \(T_{e/f}\) denote the swap tree obtained from \(T\) by replacing \(e\) with \(f\).

Given two vertices \(x, y \in V(G)\), we denote by \(d_G(x, y)\) the distance between \(x\) and \(y\) in \(G\), i.e., the number of edges contained in a shortest path in \(G\) between \(x\) and \(y\). We define the stretch factor \(\sigma_G(T)\) of \(T\) w.r.t. \(G\) as

\[
\sigma_G(T) = \max_{x, y \in V(G)} \frac{d_T(x, y)}{d_G(x, y)}.
\]

**Definition 1 (Best Swap Edge).** Let \(e \in E(T)\). An edge \(f^* \in S(e)\) is a best swap edge for \(e\) if \(f^* \in \arg\min_{f \in S(e)} \sigma_{G-e}(T_{e/f})\).

For a rooted tree \(T\) and two vertices \(u\) and \(v\) of \(T\), we denote by \(A(v)\) the set of all the proper ancestors of \(v\) in \(T\), we denote by \(p(v)\) the parent of \(v\) in \(T\), and we denote by \(lca(u, v)\) the least common ancestor of \(u\) and \(v\) in \(T\).

3 The algorithm

In this section we design an \(O(n^2)\) time and space algorithm that computes a best swap edge for every edge of \(T\). Let \(r\) be an arbitrarily chosen vertex of \(T\). For the rest of the paper, we assume that \(T\) is rooted at \(r\). The algorithm works as follows. First, for every vertex \(x\) of \(T\), the algorithm computes the set \(E(x) := \{(x, y) \in E(G) \setminus E(T) \mid x \notin A(y)\}\) of non-tree edges of the form \((x, y)\), where \(x\) is not an ancestor of \(y\) in \(T\) (see Figure 1). Observe that some sets \(E(x)\) may be empty. Observe also that each edge \((x, y)\) such that \(x \notin A(y)\) and \(y \notin A(x)\) is contained in both \(E(x)\) and \(E(y)\). The precomputation of all the sets \(E(x)\) requires linear time if we use a data structure that can compute the least common ancestor of any 2 given vertices in constant time [11].

The algorithm visits the edges of \(T\) in postorder and, for each edge \(e \in E(T)\), it computes a corresponding best swap edge in \(O(n)\) time. For the rest of the paper, unless stated otherwise, let \(e = (p(v), v)\) be a fixed tree edge and let \(X\) be the set of vertices contained in the subtree of \(T\) rooted at \(v\). The algorithm computes a best swap edge \(f^*\) of \(e\) as follows.
Figure 1 An example showing how the set $E(x)$ is defined. Tree edges are solid, while swap edges are dashed. In this example $E(x) = \{ f_1, f_2, f_3 \}$.

First, for every $x \in X$, the algorithm computes a candidate best swap edge $f_x$ of $e$ that is chosen among the edges of $F(x, e) := E(x) \cap S(e)$. More precisely,

$$f_x \in \arg \min_{f \in F(x, e)} \sigma_{G-e}(T_{e/f}).$$

The best swap edge $f^*$ is then selected among the computed candidate best swap edges. More precisely,

$$f^* \in \arg \min_{f_x \in F(x, e)} \min_{x \in X} \sigma_{G-e}(T_{e/f_x}). \tag{1}$$

We can prove the following lemma.

Lemma 2. The edge $f^*$ computed as in (1) is a best swap edge of $e$.

Proof. Let $x \in X$ and let $(x, y) \in S(e)$ be any swap edge of $e$ incident to $x$. Since $x \not\in A(y)$, we have that $(x, y) \in E(x)$. Therefore, $(x, y) \in F(x, e)$. As a consequence, $S(e) = \bigcup_{x \in X} F(x, e)$.

Hence

$$\sigma_{G-e}(T_{e/f^*}) = \min_{f_x \in F(x, e)} \min_{x \in X} \sigma_{G-e}(T_{e/f_x}) = \min_{x \in X} \sigma_{G-e}(T_{e/f_x}) \min_{f \in S(e)} \sigma_{G-e}(T_{e/f}).$$

The claim follows.

3.1 How to compute the candidate best swap edges

As already proved in Lemma 3 of [2], the candidate best swap edge $f_x$ can be computed via a reduction to the subset minimum eccentricity problem on trees. We revise the reduction in the following. In the subset minimum eccentricity problem on trees, we are given a tree $T$, with a cost $c(y)$ associated with each vertex $y$, and a subset $Y \subseteq V(T)$, and we are asked to find a vertex in $Y$ of minimum eccentricity, i.e., a vertex $y^* \in Y$ such that

$$y^* \in \arg \min_{y \in Y} \max_{y' \in V(T)} \left( d_T(y, y') + c(y') \right).$$

The reduction from the problem of computing the candidate best swap edge $f_x$ to the subset minimum eccentricity problem on trees is as follows. The input tree corresponds to $T$, the cost associated with each vertex $y$ is $c_y := \max_{x' \in X, (x', y) \in S(e)} d_T(x', x)$, and the subset of vertices from which we have to choose the one with minimum eccentricity is $Y(x, e) := \{ y \mid (x, y) \in F(x, e) \}$. As the following lemma shows, the problem can be solved by computing:

1. With a little abuse of notation, if $F(x, e) = \emptyset$, then $f_x = \perp$ and $\sigma_{G-e}(T_{e/f_x}) = +\infty$.
2. If $S(e)$ contains no edge incident to $y$, then $c_y = -\infty$. 
the endvertices of a *diametral path* of $T$, i.e., two (not necessarily distinct) vertices $a_x, b_x \in V(T)$ such that

$$\{a_x, b_x\} \in \arg \max_{\{a,b\},a,b \in V(T)} (c_x(a) + d_T(a, b) + c_x(b));$$

- a *center* of $T$, i.e., a vertex $\gamma_x \in V(T)$ such that

$$\gamma_x \in \arg \min_{\gamma \in V(T)} \max_{y \in V(T)} (d_T(\gamma, y) + c_x(y)).$$

**Lemma 3** (Bilò et al. [2], Lemma 6 and Lemma 7). Let $\gamma_x$ be a center of $T$ and let $a_x$ and $b_x$ be the two endvertices of a diametral path $P$ in $T$. Then $\gamma_x$ is also a center of $P$. Furthermore, if $y_x \in Y(x, c)$ is the vertex closest to the center $\gamma_x$, i.e., $y_x \in \arg \min_{y \in Y(x, c)} d_T(y, \gamma_x)$, then $f_x := (x, y_x)$ is a candidate best swap edge of $c$ and $\sigma_{G-e}(T_{e/f_x}) = 1 + \max \{d_T(y_x, a_x) + c_x(a_x), d_T(y_x, b_x) + c_x(b_x)\}$.

In what follows we show how all the vertices $y_x$ and all the values $\sigma_{G-e}(T_{e/f_x})$, for every $x \in X$, can be computed in $O(n)$ time and space. More precisely, the algorithm first computes the endvertices $a_x$ and $b_x$, for every $x \in X$, in $O(n)$ time and space. Thanks to Lemma 3, once both $a_x$ and $b_x$ are known, and since all tree edges have length equal to 1, we can compute $\gamma_x$ in constant time using a constant number of least common ancestor and level ancestor queries [1, 3]. Finally, for each $x \in X$, we show how to compute the vertex $y_x$ that is closest to $\gamma_x$ in constant time using range-minimum-query data structures [1, 11].

### 3.1.1 How to compute the endvertices of the diametral paths

To compute $a_x$ and $b_x$, we make use of the following key lemma.

**Lemma 4** (Merge diameter lemma). Let $T$ be a tree, with a cost $c(y)$ associated with each $y \in V(T)$. Let $c_1, \ldots, c_\ell$ be $\ell$ (vertex-cost) functions and let $k_1, \ldots, k_\ell$ be $\ell$ constants such that, for every vertex $y \in V(T)$, $c(y) = \max_{i=1,\ldots,\ell} (c_i(y) + k_i).$ For every $i = 1, \ldots, \ell$, let $a_i, b_i$ be the two endvertices of a diametral path of $T$ w.r.t. the cost function $c_i$. Then, there are two indices $i, j = 1, \ldots, \ell$ (i may also be equal to j) and two vertices $a \in \{a_i, b_i\}$ and $b \in \{a_j, b_j\}$ such that:

1. $a$ and $b$ are the two endvertices of a diametral path of $T$ w.r.t. cost function $c$;
2. $c(a) = c_i(a) + k_i$;
3. $c(b) = c_j(b) + k_j$.

Furthermore, if $a_i, b_i$, and their corresponding costs $c_i(a_i)$ and $c_i(b_i)$ are known for every $i = 1, \ldots, \ell$, then the vertices $a$ and $b$ can be computed in $O(\ell)$ time and space.

**Proof.** Let $a, b$ be the two endvertices of a diametral path in $T$ w.r.t. the cost function $c$. For some $i, j = 1, \ldots, \ell$, we have that $c(a) = c_i(a) + k_i$ as well as $c(b) = c_j(b) + k_j$ (i may also be equal to j). Let $P_1$ (resp., $P_2$) be the path in $T$ between $a_i$ (resp., $a_j$) and $b_i$ (resp., $b_j$). Let $t$ be the first vertex of the path in $T$ from $a$ to $a_i$ that is also in $P_1$, where we assume that the path is traversed in the direction from $a$ to $a_i$. Similarly, let $t'$ be the first vertex of the path in $T$ from $b$ to $b_j$ that is also in $P_2$, where we assume that the path is traversed in the direction from $b$ to $b_j$. We claim that there are $\bar{a} \in \{a_i, b_i\}$ and $\bar{b} \in \{a_j, b_j\}$ such that

$$d_T(\bar{a}, \bar{b}) = d_T(\bar{a}, t) + d_T(t, t') + d_T(t', \bar{b}).$$

\[\text{(2)}\]

---

3 Indeed, by computing the least common ancestor between $a_x$ and $b_x$, say $\tilde{x}$, we know whether $\gamma_x$ is along either the $\tilde{x}$ to $a_x$ path or the $\tilde{x}$ to $b_x$ path. If $\gamma_x$ is an ancestor of $a_x$, then its distance from $a_x$ is equal to $\frac{[c_{\tilde{x}}(a_x) + d_T(a_x, b_x) + c_{\tilde{x}}(b_x)]/2 - c_{\tilde{x}}(a_x)}$. If $\gamma_x$ is an ancestor of $b_x$, then its distance from $b_x$ is equal to $\frac{[c_{\tilde{x}}(a_x) + d_T(a_x, b_x) + c_{\tilde{x}}(b_x)]/2 - c_{\tilde{x}}(b_x)}$.\]
Indeed, we observe that at least one of the two paths in $T$ from $a_i$ to $t'$ and from $b_i$ to $t'$ passes through $t$. W.l.o.g., we assume that the path in $T$ from $a_i$ to $t'$ passes through $t$ (see Figure 2). Similarly, at least one of the two paths in $T$ from $a_i$ to $a_j$ and from $a_i$ to $b_j$ passes through $t'$. As a consequence, such a path also passes through $t$. Therefore $\bar{a} = a_i$ and $\bar{b} = a_j$ (see Figure 2).

Let $\bar{b} \in \{a_i, b_i\}$, with $\bar{b} \neq \bar{a}$, and $\bar{a} \in \{a_j, b_j\}$, with $\bar{a} \neq \bar{b}$. Since $\bar{a}$ and $\bar{b}$ are the endvertices of a diametral path in $T$ w.r.t. the cost function $c_i$, we have that

$$c_i(a) + d_T(a, t) + d_T(t, \bar{b}) + c_i(\bar{b}) = c_i(a) + d_T(a, \bar{b}) + c_i(\bar{b})$$

$$\leq c_i(\bar{a}) + d_T(\bar{a}, \bar{b}) + c_i(\bar{b})$$

$$= c_i(\bar{a}) + d_T(\bar{a}, t) + d_T(t, \bar{b}) + c_i(\bar{b}),$$

from which we derive

$$c_i(a) + d_T(a, t) \leq c_i(\bar{a}) + d_T(\bar{a}, t). \tag{3}$$

Similarly, since $\bar{a}$ and $\bar{b}$ are the endvertices of a diametral path in $T$ w.r.t. the cost function $c_j$, we have that

$$c_j(\bar{a}) + d_T(\bar{a}, t') + d_T(t', b) + c_j(b) = c_j(\bar{a}) + d_T(\bar{a}, b) + c_j(b)$$

$$\leq c_j(\bar{a}) + d_T(\bar{a}, \bar{b}) + c_j(\bar{b})$$

$$= c_j(\bar{a}) + d_T(\bar{a}, t') + d_T(t', \bar{b}) + c_j(\bar{b}),$$

from which we derive

$$d_T(t', b) + c_j(b) \leq d_T(t', \bar{b}) + c_j(\bar{b}). \tag{4}$$

Using Inequality (3) and Inequality (4), together with Equality (2), we obtain

$$c(a) + d_T(a, b) + c(b) \leq c(a) + d_T(a, t) + d_T(t, t') + d_T(t', b) + c(b)$$

$$= c_i(a) + k_i + d_T(a, t) + d_T(t, t') + d_T(t', b) + c_j(b) + k_j$$

$$\leq c_i(\bar{a}) + k_i + d_T(\bar{a}, t) + d_T(t, t') + d_T(t', \bar{b}) + c_j(\bar{b}) + k_j$$

$$= c_i(\bar{a}) + k_i + d_T(\bar{a}, \bar{b}) + c_j(\bar{b}) + k_j$$

$$\leq c(\bar{a}) + d_T(\bar{a}, \bar{b}) + c(\bar{b}).$$

Since $a$ and $b$ are the two endvertices of a diametral path in $T$ w.r.t. cost function $c$, the above inequality is satisfied at equality. As a consequence, $a = \bar{a}$ and $b = \bar{b}$ satisfy all the three conditions of the lemma statement.
We complete the proof by showing that $a$ and $b$ can be computed in $O(\ell)$ time using dynamic programming. For every $i = 1, \ldots, \ell$, we compute the endvertices $\alpha_i$ and $\beta_i$ of a diametral path in $T$ w.r.t. the cost function $\psi_i := \max_{1 \leq j \leq i} (c_j(y) + k_j)$, together with their corresponding costs. Clearly, for $i = 1$, $\alpha_1 = a_1$, $\beta_1 = b_1$, $\psi_1(\alpha_1) = c_1(a_1) + k_1$, and $\psi_1(\beta_1) = c_1(b_1) + k_1$. Moreover, for every $i \geq 2$, we can compute $\alpha_i$ and $\beta_i$, together with $\psi_i(\alpha_i)$ and $\psi_i(\beta_i)$, in constant time and space from $\alpha_{i-1}$ and $\beta_{i-1}$, where $\psi_i(x) = \psi_{i-1}(x)$ for $x \in \{\alpha_{i-1}, \beta_{i-1}\}$, and from $a_i$ and $b_i$, where $\psi_i(x) = c_i(x) + k_i$ for $x \in \{a_i, b_i\}$. Therefore, $\alpha_\ell$ and $\beta_\ell$, together with $\psi_\ell(\alpha_\ell)$ and $\psi_\ell(\beta_\ell)$, can be computed in $O(\ell)$ time and space. The claim follows by observing that $a = \alpha_\ell$ and $b = \beta_\ell$.

Lemma 5. For every $x \in V(T)$ and every $z \in A(x)$, all the vertices $a_{x|z}, b_{x|z}$ and their corresponding costs w.r.t. $c_{x|z}$ can be computed in $O(n^2)$ time and space.

Proof. We show that, for any $x \in V(T)$ and any $z \in A(x)$, the vertices $a_{x|z}$ and $b_{x|z}$ can be computed in $O(1 + |Y(x|z)|)$ time and space. The claim would follow immediately since

$$\sum_{x \in V(T)} \sum_{z \in A(x)} O\left(1 + |Y(x|z)|\right) = \sum_{x \in V(T)} O\left(\sum_{z \in A(x)} (1 + |Y(x|z)|)\right) = \sum_{x \in V(T)} O(n) = O(n^2).$$

Let $x \in V(T)$ and $z \in A(x)$ be fixed, and let $\ell = |Y(x|z)|$. Let $y_1, \ldots, y_\ell$ be the $\ell$ vertices of $Y(x|z)$ and, finally, for every $i = 1, \ldots, \ell$, let

$$c_i(y) := \begin{cases} 0 & \text{if } y = y_i; \\ -\infty & \text{otherwise}. \end{cases}$$

We have that $a_i = b_i = y_i$ are the two endvertices of the unique diametral path in $T$ w.r.t. cost function $c_i$. Moreover, for every $y \in V(T)$, we have that $c_{x|z}(y) = \max_{1 \leq i \leq \ell} c_i(y)$. Therefore, using Lemma 4, we can compute $a_{x|z}, b_{x|z}$, and their corresponding costs w.r.t. $c_{x|z}$, in $O(1 + |Y(x|z)|)$ time and space.\hfill\blacktriangleup

Let $c_{x,e}$ be a cost function that, for every $y \in V(T)$, is defined as follows $c_{x,e}(y) := \max_{z \in A(e)} c_{z|y}$. The algorithm also precomputes the two endvertices $a_{x,e}$ and $b_{x,e}$ of a diametral path of $T$ w.r.t. the cost function $c_{x,e}$, together with the corresponding values $c_{x,e}(a_{x,e})$ and $c_{x,e}(b_{x,e})$. The following lemma holds.

Lemma 6. For every $x \in V(T)$ and every edge $e$ in the path between $r$ and $x$, all the vertices $a_{x,e}, b_{x,e}$ and their corresponding costs w.r.t. $c_{x,e}$ can be computed in $O(n^2)$ time and space.

Proof. Let $x \in V(T)$ and let $e$ be an edge of the path between $r$ and $x$ in $T$. We show that $a_{x,e}, b_{x,e}$ (and their corresponding costs w.r.t. $c_{x,e}$) can be computed in constant time and space. The claim then follows since $V(T), E(T) = O(n)$. We divide the proof into two cases.
The first case occurs when \( e \) is incident to \( r \). Clearly, for every \( y \in V(T) \), \( c_{x,e}(y) = c_{x,r}(y) \).

As a consequence, \( a_{x,e} = a_{x,r} \) and \( b_{x,e} = b_{x,r} \).

The second case occurs when \( e = (u = p(v), v) \), with \( u \neq r \). Let \( e' = (p(u), u) \). Then, for every \( y \in V(T) \), \( c_{x,e}(y) = \max \{ c_{x,e'}(y), c_{x,u}(y) \} \).

Therefore, using Lemma 4, for every \( x \), \( c_{x,e}(y) \) can be computed in \( O(n^2) \) time and space via a preorder visit of the tree edges.

In the following we show how to compute \( a_x \) and \( b_x \), for every \( x \in X \), in \( O(n) \) time and space. First, for every \( x \in X \), we consider a subdivision of \( X \) into three sets \( Z(x,1), Z(x,2), \) and \( Z(x,3) \) (see Figure 3) such that:

- \( Z(x,1) \) is the set of all the descendants of \( x \) in \( T \) (\( x \) included);
- \( Z(x,2) \) is the union of all the sets \( Z(s,1) \), for every sibling \( s \) of \( x \);
- \( Z(x,3) = X \setminus (Z(x,1) \cup Z(x,2)) \).

Each set \( Z(x,i) \) is associated with a cost function \( c_{x,i} \) that, for every \( y \in V(T) \), is defined as follows:

\[
c_{x,i}(y) := \max_{x' \in Z(x,i),(x',y) \in S(e)} d_T(x',x).
\]

Let \( a_{x,i} \) and \( b_{x,i} \) be the two endvertices of a diametral path in \( T \) w.r.t. cost function \( c_{x,i} \). The algorithm computes all the vertices \( a_{x,i}, b_{x,i} \) and their corresponding values w.r.t. cost function \( c_{x,i} \), for every \( x \in X \) and every \( i = 1, 2, 3 \).

**Lemma 7.** For every \( x \in X \) and every \( i \in \{1, 2, 3\} \), all the vertices \( a_{x,i}, b_{x,i} \) and their corresponding costs w.r.t. \( c_{x,i} \) can be computed in \( O(n) \) time and space.

**Proof.** We divide the proof into three cases, according to the value of \( i \).

The first case is \( i = 1 \). Clearly, if \( x \) is a leaf vertex, then \( a_{x,1} = a_{x,e} \) and \( b_{x,1} = b_{x,e} \). Moreover, \( c_{x,1}(a_{x,1}) = c_{x,e}(a_{x,e}) \) as well as \( c_{x,1}(b_{x,1}) = c_{x,e}(b_{x,e}) \). Therefore, we can assume that \( x \) is not a leaf vertex. Let \( x_1, \ldots, x_{\ell-1} \) be the \( \ell - 1 \) children of \( x \) in \( T \). Since \( Z(x,1) = \{ x \} \cup \bigcup_{i=1}^{\ell-1} Z(x_i,1) \), for every \( y \in V(T) \), we have that

\[
c_{x,1}(y) = \max \left\{ c_{x,e}(y), 1 + \max_{i=1,\ldots,\ell-1} c_{x_i,1}(y) \right\}.
\]

Therefore, using Lemma 4, for every \( x \in X \), all the vertices \( a_{x,1}, b_{x,1} \), together with their corresponding costs w.r.t. \( c_{x,i} \), can be computed in \( O(n) \) time and space via a postorder visit of the vertices in \( X \).
We consider the case in which \( i = 2 \) and we assume that, for every \( x \in X \), all the vertices \( a_{x,1}, b_{x,1} \) and their corresponding costs w.r.t. \( c_{x,1} \) are known. Let \( \bar{x} \) be the parent of \( x \) in \( T \) and let \( x_1, \ldots, x_\ell \) be the \( \ell \geq 1 \) children of \( \bar{x} \) in \( T \). For every \( i = 1, \ldots, \ell \), let \( \bar{c}_{x,i} \) and \( \bar{c}_{x,1} \) be the two cost functions that, for every \( y \in V(T) \), are defined as follows:

\[
\bar{c}_{x,i}(y) := 2 + \max_{j=1, \ldots, i-1} c_{x,j,1}(y),
\]

\[
\bar{c}_{x,i}(y) := 2 + \max_{j=i+1, \ldots, \ell} c_{x,j,1}(y).
\]

For every \( i = 2, \ldots, \ell - 1 \), the algorithm computes the vertices \( \bar{a}_{x,i}, \bar{b}_{x,i}, \bar{a}_{x,i}, \bar{b}_{x,i} \) and the costs \( \bar{c}_{x,i}(\bar{a}_{x,i}), \bar{c}_{x,i}(\bar{b}_{x,i}), \bar{c}_{x,i}(\bar{a}_{x,i}), \bar{c}_{x,i}(\bar{b}_{x,i}) \) using simple dynamic programming and Lemma 4. Indeed, \( \bar{c}_{x,2}(y) = 2 + c_{x,1,1}(y) \) as well as \( \bar{c}_{x,\ell-1}(y) = 2 + c_{x,1,1}(y) \). Furthermore, for every \( i > 2 \),

\[
\bar{c}_{x,i}(y) = \max \{ \bar{c}_{x,i-1}(y), 2 + c_{x,i-1,1}(y) \},
\]

while for every \( i < \ell - 1 \),

\[
\bar{c}_{x,i}(y) = \max \{ \bar{c}_{x,i+1}(y), 2 + c_{x,i+1,1}(y) \}.
\]

We observe that \( \bar{a}_{x,i}, \bar{b}_{x,i}, \bar{a}_{x,i}, \bar{b}_{x,i} \) and the costs \( \bar{c}_{x,i}(\bar{a}_{x,i}), \bar{c}_{x,i}(\bar{b}_{x,i}), \bar{c}_{x,i}(\bar{a}_{x,i}), \bar{c}_{x,i}(\bar{b}_{x,i}) \) can be computed in \( O(\ell) \) time and space. As a consequence, these pieces of information can be precomputed for every \( x \in V(T) \) in \( O(n^2) \) time and space. Let \( x = x_i \), for some \( i = 1, \ldots, \ell \). Since

\[
Z(x, 2) = \bigcup_{j=1, \ldots, \ell, j \neq i} Z(x_j, 1) = \bigcup_{j=1}^{i-1} Z(x_j, 1) \cup \bigcup_{j=i+1}^{\ell} Z(x_j, 1),
\]

for every \( y \in V(T) \), we have that

\[
c_{x,2}(y) = \max \{ \bar{c}_{x,i}(y), \bar{c}_{x,i}(y) \}.
\]

Therefore, using Lemma 4, all the vertices \( a_{x,2}, b_{x,2} \) and their corresponding costs w.r.t. \( c_{x,2} \) can be computed in \( O(n) \) time and space for every \( x \in X \).

Finally, we consider the case in which \( i = 3 \) and we assume that all the vertices \( a_{x,j}, b_{x,j} \) and the costs \( c_{x,j}(a_{x,j}), c_{x,j}(b_{x,j}) \), with \( x \in X \) and \( j = 1, 2 \), are known. If \( x = v \), then \( Z(x, 3) = \emptyset \). Therefore, we only need to prove the claim when \( x \neq v \). Let \( \bar{x} \) be the parent of \( x \) in \( T \). Since

\[
Z(x, 3) = \{ \bar{x} \} \cup Z(\bar{x}, 2) \cup Z(\bar{x}, 3)
\]

for every \( y \in V(T) \), we have that

\[
c_{x,3}(y) = 1 + \max \{ \bar{c}_{x,e}(y), c_{x,2}(y), \bar{c}_{x,2}(y) \}.
\]

Therefore, using Lemma 4, for every \( x \in X \), all the vertices \( a_{x,3}, b_{x,3} \), and the corresponding costs w.r.t. \( c_{x,3} \), can be computed in \( O(n) \) time and space by a preorder visit of the tree vertices. This completes the proof.

We can now prove the following.

**Lemma 8.** For every \( x \in X \), all the vertices \( a_x, b_x, \gamma_x \) and the costs \( c_x(a_x) \) and \( c_x(b_x) \) can be computed in \( O(n) \) time and space.
A Novel Algorithm for the ABSE Problem on Tree Spanners

**Proof.** Let \( x \in X \) be fixed. Since \( X = Z(x,1) \cup Z(x,2) \cup Z(x,3) \), by definition of \( c_{x,i} \), for every \( y \in V(T) \), we have that \( c_x(y) = \max \{ c_{x,i}(y) : i = 1,2,3 \} \). Therefore, under the assumption that all the vertices \( a_{x,i}, b_{x,i} \) and all the values \( c_{x,i}(a_{x,i}), c_{x,i}(b_{x,i}) \), with \( x \in X \) and \( i = 1,2,3 \), are known, using Lemma 4, we can compute \( a_x \) and \( b_x \), together with the values \( c_x(a_x) \) and \( c_x(b_x) \) in constant time. The claim follows since, as we already discussed at the end of Section 3.1, \( \gamma_x \) can be computed in constant time.

\[ \square \]

### 3.2 How to compute the vertex \( y_x \)

**Lemma 9.** The labeling \( \lambda \) and the two range-minimum-query data structures \( R \) and \( R' \) can be computed in \( O(n) \) time and space.

**Proof.** It is known that the range-minimum-query data structure of size \( h \) can be computed in \( O(h) \) time and space [1, 11]. The labeling \( \lambda \) can be computed in \( O(n) \) time and space by a simple algorithm that, for every \( i = 1, \ldots, h \), first initializes \( \lambda(y) = y \) for every \( y \in Y(x, z_i) \) and then, during phase \( \phi \), labels all the still unlabeled vertices which are at distance \( \phi \) from some vertex in \( Y(x, z_i) \).

Let \( e = (z_i = p(v), v) \) be the failing edge. We show how to find the vertex \( y_x \in Y(x, e) \) that is closest to \( \gamma_x \) in constant time. First we compute \( j \) such that \( z_j = lca(\gamma_x, x) \). Next we make at most two range minimum queries to compute the following two indices:

- The index \( t' \) containing the minimum value within the range \([1, j-1]\) in \( R'\);
- The index \( t \) containing the minimum value within the range \([j+1, i]\) in \( R \).

The algorithm chooses \( y_x \) such that

\[
y_x \in \arg \min_{y \in \{\lambda(\gamma_x), \lambda(z_j), \lambda(z_i)\}} d_T(y, \gamma_x),
\]

**Lemma 10.** The vertex \( y_x \) selected by the algorithm satisfies \( y_x \in \arg \min_{y \in Y(x,e)} d_T(y, \gamma_x) \).

**Proof.** Let \( y^* \in \arg \min_{y \in Y(x,e)} d_T(y, \gamma_x) \). Clearly, for some \( k = 1, \ldots, i \), \( y^* \in Y(x, z_k) \). We prove the claim by showing that \( d_T(y_x, \gamma_x) \leq d_T(y^*, \gamma_x) \). We divide the proof into three cases, according to the value of \( k \).

The first case is when \( k = j \). We have that \( d_T(y_x, \gamma_x) \leq d_T(\lambda(\gamma_x), \gamma_x) = d_T(y^*, \gamma_x) \).

The second case occurs when \( k < j \). Clearly, \( d_T(\lambda(z_k), z_k) \leq d_T(y^*, z_k) \). Moreover, \( d_T(\lambda(z_j), z_j) - t' \leq d_T(\lambda(z_k), z_k) - k \). Therefore,

\[
d_T(y_x, \gamma_x) \leq d_T(\lambda(z_k), \gamma_x) = d_T(\lambda(z_j), z_j) + d_T(z_j, \gamma_x) - t' + d_T(y^*, z_k) \leq d_T(\lambda(z_k), z_k) + j - k + d_T(\lambda(z_j), \gamma_x)
\]

The third case occurs when \( j < k \). Clearly, \( d_T(\lambda(z_k), z_k) \leq d_T(\gamma_x, z_k) \). Moreover, \( d_T(\lambda(z_i), z_i) + t \leq d_T(\lambda(z_k), z_k) + k \). Therefore,

\[
d_T(y_x, \gamma_x) \leq d_T(\lambda(z_i), \gamma_x) = d_T(\lambda(z_k), z_k) + d_T(z_k, \gamma_x) - j + d_T(z_j, \gamma_x) \]

The claim follows.

We can finally state the main theorem.
Theorem 11. All the best swap edges of a tree spanner $T$ in 2-edge-connected, unweighted, and undirected graphs can be computed in $O(n^2)$ time and space.

Proof. From Lemma 8, for a fixed edge $e \in E(T)$, all the vertices $a_x, b_x, \gamma_x$ and all the values $c_x(a_x), c_x(b_x)$, with $x \in X$, can be computed in $O(n)$ time and space. Therefore, such vertices and values can be computed for every edge of $T$ in $O(n^2)$ time and space.

By Lemma 10, for a fixed edge $e$ of $T$ and a fixed vertex $x$, we can compute $y_x$, i.e., $f_x = (x, y_x)$, by making at most two queries, each of which requires constant time, on the two range-minimum-query data structures associated with $x$. Therefore, the $O(n)$ candidate best swap edges of $e$ can be computed in $O(n)$ time. Furthermore, using Lemma 3, we can compute $\sigma(T_{e/f_x}) = 1 + \max \{d_T(y_x, a_x) + c_x(a_x), d_T(y_x, b_x) + c_x(b_x)\}$ in constant time. Hence, thanks to Lemma 2, the best swap edge $f^*$ of $e$ can be computed in $O(n)$ time. The claim follows.

References


A Novel Algorithm for the ABSE Problem on Tree Spanners


Efficient Enumeration of Dominating Sets for Sparse Graphs

Kazuhiro Kurita
IST, Hokkaido University, Sapporo, Japan
k-kurita@ist.hokudai.ac.jp

Kunihiro Wasa
National Institute of Informatics, Tokyo, Japan
wasa@nii.ac.jp

Hiroki Arimura
IST, Hokkaido University, Sapporo, Japan
arim@ist.hokudai.ac.jp

Takeaki Uno
National Institute of Informatics, Tokyo, Japan
uno@nii.ac.jp

Abstract
A dominating set $D$ of a graph $G$ is a set of vertices such that any vertex in $G$ is in $D$ or its neighbor is in $D$. Enumeration of minimal dominating sets in a graph is one of central problems in enumeration study since enumeration of minimal dominating sets corresponds to enumeration of minimal hypergraph transversal. However, enumeration of dominating sets including non-minimal ones has not been received much attention. In this paper, we address enumeration problems for dominating sets from sparse graphs which are degenerate graphs and graphs with large girth, and we propose two algorithms for solving the problems. The first algorithm enumerates all the dominating sets for a $k$-degenerate graph in $O(k)$ time per solution using $O(n + m)$ space, where $n$ and $m$ are respectively the number of vertices and edges in an input graph. That is, the algorithm is optimal for graphs with constant degeneracy such as trees, planar graphs, $H$-minor free graphs with some fixed $H$. The second algorithm enumerates all the dominating sets in constant time per solution for input graphs with girth at least nine.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases Enumeration algorithm, polynomial amortized time, dominating set, girth, degeneracy

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.8

1 Introduction

One of the fundamental tasks in computer science is to enumerate all subgraphs satisfying a given constraint such as cliques [23], spanning trees [25], cycles [2], and so on. One of the approaches to solve enumeration problems is to design exact exponential algorithms, i.e., input-sensitive algorithms. Another mainstream of solving enumeration problems is to design output-sensitive algorithms, i.e., the computation time depends on the sizes of both of an input and an output. An algorithm $A$ is output-polynomial if the total computation time is polynomial of the sizes of input and output. $A$ is an incremental polynomial time algorithm if the algorithm needs $O(poly(n, i))$ time when the algorithm outputs the $i$th solution after outputting the $(i - 1)$th solution, where $poly(\cdot)$ is a polynomial function. $A$
runs in \( \text{polynomial amortized time} \) if the total computation time is \( O(poly(n)N) \), where \( n \) and \( N \) are respectively the sizes of an input and an output. In addition, \( \mathcal{A} \) runs in \( \text{polynomial delay} \) if the maximum interval between two consecutive solutions is \( O(poly(n)) \) time and the preprocessing and postprocessing time is \( O(poly(n)) \). From the point of view of tractability, efficient algorithms for enumeration problems have been widely studied \([1,2,6,11,12,20,23,25,27]\). On the other hands, Lawler et al. show that some enumeration problems have no output-polynomial time algorithm unless \( P = NP \) \([21]\). In addition, recently, Creignou et al. show a tool for showing the hardness of enumeration problems \([8]\).

A dominating set is one of a fundamental substructure of graphs and finding the minimum dominating set problem is a classical NP-hard problem \([12]\). A vertex set \( D \) of a graph \( G \) is a dominating set of \( G \) if every vertex in \( G \) is in \( D \) or has at least one neighbors in \( D \). The enumeration of minimal dominating sets of a graph is closely related to the enumeration of minimal hypergraph transversals of a hypergraph \([10]\). Kanté et al. \([18]\) show that the minimal dominating set enumeration problem and the minimal hypergraph transversal enumeration problem are equivalent, that is, the one side can be solved in output-polynomial time if the other side can be also solved in output-polynomial time. Several algorithms that run in polynomial delay have been developed when we restrict input graphs, such as permutation graphs \([18]\), chordal graphs \([19]\), line graphs \([20]\), graphs with bounded degeneracy \([16]\), graphs with bounded tree-width \([7]\), graphs with bounded clique-width \([7]\), and graphs with bounded (local) LMIM-width \([14]\). Incremental polynomial-time algorithms have also been developed, such as chordal bipartite graphs \([13]\), graphs with bounded conformality \([3]\), and graphs with girth at least seven \([15]\). Kanté et al. \([17]\) show that the conformality of the closed neighbourhood hypergraphs of line graphs, path graphs, and \((C_4, C_5, claw)-free\) graphs is constant. However, it is still open whether there exists an output-polynomial time algorithm for enumerating minimal dominating sets from general graphs.

Since the number of solutions exponentially increases compared to the minimal version, even if we can develop an enumeration algorithm that runs in constant time per solution, the algorithm becomes theoretically much slower than some enumeration algorithm for minimal dominating sets. However, when we consider the real-world problem, we sometimes use another criteria for enumerating solutions that form dominating sets in a graph. That is, enumeration algorithms for minimal dominating sets may not fit in with other variations of minimal domination problems. E.g., a tropical dominating set \([9]\) and a rainbow dominating set \([4]\) are such a dominating set. Thus, when we enumerate solutions of such domination problems, our algorithm becomes a base-line algorithm for these problems. Thus, our main goal is to develop an efficient enumeration algorithm for dominating sets.

**Main results:** In this paper, we consider the relaxed problems, i.e., enumeration of all dominating sets that include non-minimal ones in a graph. We present two algorithms, \( \text{EDS-D} \) and \( \text{EDS-G} \). \( \text{EDS-D} \) enumerates all dominating sets in \( O(k) \) time per solution, where \( k \) is the degeneracy of a graph (Theorem 13). Moreover, \( \text{EDS-G} \) enumerates all dominating sets in constant time per solution for a graph with girth at least nine (Theorem 25), where the girth is the length of minimum cycle in the graph.

By straightforwardly using an enumeration framework such as the reverse search technique \([1]\), we can obtain an enumeration algorithm for the problem that runs in \( O(n) \) or \( O(\Delta) \) time per solution, where \( n \) and \( \Delta \) are respectively the number of vertices and the maximum degree of an input graph. Although dominating sets are fundamental in computer science, no enumeration algorithm for dominating sets that runs in strictly faster than such a trivial algorithm has been developed so far. Thus, to develop efficient algorithms, we focus
on the sparsity of graphs as being a good structural property and, in particular, on the degeneracy and girth, which are the measures of sparseness. As our contributions, we develop two optimal algorithms for enumeration of dominating sets in a sparse graph. We first focus on the degeneracy of an input graph. A graph is \( k \)-degenerate [22] if any subgraph of the graph has a vertex whose degree is at most \( k \). The degeneracy of a graph is the minimum value of \( k \) such that the graph is \( k \)-degenerate. Note that \( k \leq \Delta \) always holds. It is known that some graph classes have constant degeneracy, such as forests, grid graphs, outerplanar graphs, planar graphs, bounded tree width graphs, and \( H \)-minor free graphs for some fixed \( H \) [5, 26]. A \( k \)-degenerate graph has a good vertex ordering, called a degeneracy ordering [24], as shown in Section 3. So far, this ordering has been used to develop efficient enumeration algorithms [6, 11, 27]. By using this ordering and the reverse search technique [1], we show that our proposed algorithm \( \text{EDS-D} \) can solve the relaxed problem in \( O(k) \) time per solution. This implies that \( \text{EDS-D} \) can optimally enumerate all the dominating sets in an input graph with constant degeneracy.

We next focus on the girth of a graph. Enumeration of minimal dominating sets can be solved efficiently if an input graph has no short cycles since its connected subgraphs with small diameter form a tree. Indeed, this local tree structure has been used in minimal dominating sets enumeration [15]. For the relaxed problem, by using the reverse search technique, we can easily show that the delay of our proposed algorithm \( \text{EDS-G} \) for general graphs is \( O(\Delta^3) \) time. However, if an input graph has the large girth, then each recursive call generates enough solutions, that is, we can amortize the complexity of \( \text{EDS-G} \). Thus, by amortizing the time complexity using this local tree structure, we show that the problem can be solve in constant time per solution for graphs with girth at least nine.

## 2 A Basic Algorithm Based on Reverse Search

Let \( G = (V(G), E(G)) \) be a simple undirected graph, that is, \( G \) has no self loops and multiple edges, with vertex set \( V(G) \) and edge set \( E(G) \) is a set of pairs of vertices. If no confusion arises, we will write \( V = V(G) \) and \( E = E(G) \). Let \( u \) and \( v \) be vertices in \( G \). An edge \( e \) with \( u \) and \( v \) is denoted by \( e = \{u, v\} \). \( u \) and \( v \) are adjacent if \( \{u, v\} \in E \). We denote by \( N_G(u) \) the set of vertices that are adjacent to \( u \) on \( G \) and by \( N_G[u] = N_G(u) \cup \{u\} \). We say \( v \) is a neighbor of \( u \) if \( v \in N_G(u) \). The set of neighbors of \( U \) is defined as \( N(U) = \bigcup_{u \in U} N_G(u) \setminus U \). Similarly, let \( N[U] \) be \( \bigcup_{u \in U} N_G(u) \cup U \). Let \( d_G(v) = |N_G(v)| \) be the degree of \( u \) in \( G \). We call the vertex \( v \) pendant if \( d_G(v) = 1 \). \( \Delta(G) = \max_{v \in V} d(v) \) denotes the maximum degree of \( G \). A set \( X \) of vertices is a dominating set if \( X \) satisfies \( N[X] = V \).

For any vertex subset \( V' \subseteq V \), we call \( G[V'] = (V', E[V']) \) an induced subgraph of \( G \), where \( E[V'] = \{(u, v) \in E(G) \mid u, v \in V'\} \). Since \( G[V'] \) is uniquely determined by \( V' \), we identify \( G[V'] \) with \( V' \). We denote by \( G \setminus \{e\} = (V, E \setminus \{e\}) \) and \( G \setminus \{v\} = G[V \setminus \{v\}] \). For simplicity, we will use \( v \in G \) and \( e \in E \) to refer to \( v \in V(G) \) and \( e \in E(G) \), respectively.

We now define the dominating set enumeration problem as follows:

\( \textbf{Problem 1.} \) \textit{Given a graph} \( G \), \textit{then output all dominating sets in} \( G \) \textit{without duplication.} 

In this paper, we propose two algorithms EDS-D and EDS-G for solving Problem 1. These algorithms use the degeneracy ordering and the local tree structure, respectively. Before we enter into details of them, we first show the basic idea for them, called reverse search method that is proposed by Avis and Fukuda [1] and is one of the framework for constructing enumeration algorithms.

An algorithm based on reverse search method enumerates solutions by traversing on an implicit tree structure on the set of solution, called a family tree. For building the family tree,
we first define the parent-child relationship between solutions as follows: Let \( G = (V, E) \) be an input graph with \( V = \{v_1, \ldots, v_n\} \) and \( X \) and \( Y \) be dominating sets on \( G \). We arbitrarily number the vertices in \( G \) from 1 to \( n \) and call the number of a vertex the index of the vertex. If no confusion occurs, we identify a vertex with its index. We assume that there is a total ordering \( < \) on \( V \) according to the indices. \( pv (X) \), called the parent vertex, is the vertex in \( V \setminus X \) with the minimum index. For any dominating set \( X \) such that \( X \neq V \), \( Y \) is the parent of \( X \) if \( Y = X \cup \{pv (X)\} \). We denote by \( P (X) \) the parent of \( X \). Note that since any superset of a dominating set also dominates \( G \), thus, \( P (X) \) is also a dominating set of \( G \). We call \( X \) a child of \( Y \) if \( P (X) = Y \). We denote by \( F (G) \) a digraph on the set of solutions \( S (G) \). Here, the vertex set of \( F (G) \) is \( S (G) \) and the edge set \( E (G) \) of \( F (G) \) is defined according to the parent-child relationship. We call \( F (G) \) the family tree for \( G \) and call \( V \) the root of \( F (G) \). Next, we show that \( F (G) \) forms a tree rooted at \( V \).

Our basic algorithm \( EDS \) is shown in Algorithm 1. We say \( C (X) \) the candidate set of \( X \) and define \( C (X) = \{v \in V \mid N[X \setminus \{v\}] = V \land P (X \setminus \{v\}) = X\} \). Intuitively, the candidate set of \( X \) is the set of vertices such that any vertex \( v \) in the set, removing \( v \) from \( X \) generates another dominating set. We show a recursive procedure \( AllChildren(X, C (X), G) \) actually generates all children of \( X \) on \( F (G) \). We denote by \( ch(X) \) the set of children of \( X \), and by \( gch(X) \) the set of grandchildren of \( X \).

From Lemmas 1, 2, and 3, we can obtain the correctness of \( EDS \).

\textbf{Algorithm 1:} \( EDS \) enumerates all dominating sets in amortized polynomial time.

1. Procedure \( EDS(G = (V, E)) \)
   
   // \( G \): an input graph

2. \( AllChildren(V, V, G); \)

3. Procedure \( AllChildren(X, C (X), G = (V, E)) \)
   
   // \( X \): the current solution

4. Output \( X; \)

5. for \( v \in C (X) \) do

6. \( Y \leftarrow X \setminus \{v\}; \)

7. \( C (Y) \leftarrow \{u \in C (X) \mid N[Y \setminus \{u\}] = V \land P (Y \setminus \{u\}) = Y\}; \)

8. \( AllChildren(Y, C (Y), G); \)

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure.png}
\caption{An example of a degeneracy ordering for a 2-degenerate graph \( G \). In this ordering, each vertex \( v \) is adjacent to vertices at most two whose indices are larger than \( v \).}
\end{figure}

\textbf{Lemma 1.} For any dominating set \( X \), by recursively applying the parent function \( P (\cdot) \) to \( X \) at most \( n \) times, we obtain \( V \).

\textbf{Lemma 2.} \( F (G) \) forms a tree.

\textbf{Lemma 3.} Let \( X \) and \( Y \) be distinct dominating sets in a graph \( G \). \( Y \in ch(X) \) if and only if there is a vertex \( v \in C (X) \) such that \( X = Y \cup \{v\} \).

\textbf{Theorem 4.} By traversing \( F (G) \), \( EDS \) solves Problem 1.
Algorithm 2: EDS-D enumerates all dominating sets in $O(k)$ time per solution.

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Procedure EDS-D($G = (V, E)$) // $G$: an input graph</td>
</tr>
<tr>
<td>2</td>
<td>for $v \in V$ do $D_v \leftarrow \emptyset$;</td>
</tr>
<tr>
<td>3</td>
<td>AllChildren($V, V, D(V) := {D_1, \ldots, D_{</td>
</tr>
<tr>
<td>4</td>
<td>Procedure AllChildren($X, C, D$)</td>
</tr>
<tr>
<td>5</td>
<td>Output $X$;</td>
</tr>
<tr>
<td>6</td>
<td>$C' \leftarrow \emptyset; D' \leftarrow D$; // $D' := {D_1', \ldots, D_{</td>
</tr>
<tr>
<td>7</td>
<td>for $v \in C$ do // $v$ has the largest index in $C$</td>
</tr>
<tr>
<td>8</td>
<td>$Y \leftarrow X \setminus {v}$;</td>
</tr>
<tr>
<td>9</td>
<td>$C \leftarrow C \setminus {v};$ // Remove vertices in Del2($X, v$).</td>
</tr>
<tr>
<td>10</td>
<td>$C(Y) \leftarrow \text{Cand-D}(X, v, C)$; // Vertices larger than $v$ are not in $C$.</td>
</tr>
<tr>
<td>11</td>
<td>$D(Y) \leftarrow \text{DomList}(v, Y, X, C(Y), C' \oplus C(Y), D')$;</td>
</tr>
<tr>
<td>12</td>
<td>AllChildren($Y, C(Y), D(Y)$);</td>
</tr>
<tr>
<td>13</td>
<td>$C' \leftarrow C(Y)$; $D' \leftarrow D(Y)$;</td>
</tr>
<tr>
<td>14</td>
<td>for $u \in N(v)^&lt; \cup D'_u \leftarrow D'_u \cup {v}$;</td>
</tr>
<tr>
<td>15</td>
<td>Procedure Cand-D($X, v, C$)</td>
</tr>
<tr>
<td>16</td>
<td>$Y \leftarrow X \setminus {v};$ Del1 $\leftarrow \emptyset$; Del2 $\leftarrow \emptyset$;</td>
</tr>
<tr>
<td>17</td>
<td>for $u \in (N(v) \cap C) \cup N(v)^&lt;$ do</td>
</tr>
<tr>
<td>18</td>
<td>if $u &lt; v$ then</td>
</tr>
<tr>
<td>19</td>
<td>if $N(u)^&lt; \cap Y = \emptyset \land N(u)^\cap Y = \emptyset$ then Del1 $\leftarrow$ Del1 $\cup {u}$;</td>
</tr>
<tr>
<td>20</td>
<td>else</td>
</tr>
<tr>
<td>21</td>
<td>if $N[u] \cap (X \setminus C) = \emptyset \land</td>
</tr>
<tr>
<td>22</td>
<td>Return $\text{C} \setminus (\text{Del1} \cup \text{Del2})$; // $C$ is $C(X \setminus {v})$</td>
</tr>
<tr>
<td>23</td>
<td>Procedure DomList $(v, Y, X, C' \oplus C(Y), D')$</td>
</tr>
<tr>
<td>24</td>
<td>for $u \in C' \oplus C(Y)$ do</td>
</tr>
<tr>
<td>25</td>
<td>for $w \in N(u)^&lt;$ do</td>
</tr>
<tr>
<td>26</td>
<td>if $u \notin D'_w(X)$ then</td>
</tr>
<tr>
<td>27</td>
<td>if $u \notin C'$ then $D'_w \leftarrow D'_w \cup {u}$;</td>
</tr>
<tr>
<td>28</td>
<td>else $D'_w \leftarrow D'_w \setminus {u}$;</td>
</tr>
<tr>
<td>29</td>
<td>for $u \in N(v)^&lt;$ do</td>
</tr>
<tr>
<td>30</td>
<td>if $u \in X$ then $D'_u \leftarrow D'_u \cup {u}$;</td>
</tr>
<tr>
<td>31</td>
<td>return $D'$; // $D'$ is $D(Y)$</td>
</tr>
</tbody>
</table>

3 Efficient Enumeration for Bounded Degenerate Graphs

The bottleneck of EDS is the maintenance of candidate sets. Let $X$ be a dominating set and $Y$ be a child of $X$. We can easily see that the time complexity of EDS is $O(\Delta^2)$ time per solution since a removed vertex $u \in C(X) \setminus C(Y)$ has the distance at most two from $v$. In this section, we improve EDS by focusing on the degeneracy of an input graph $G$. $G$ is a $k$-degenerate graph [22] if for any induced subgraph $H$ of $G$, the minimum degree in $H$ is less than or equal to $k$. The degeneracy of $G$ is the smallest $k$ such that $G$ is $k$-degenerate. A $k$-degenerate graph has a good vertex ordering. The definition of orderings of vertices in $G$, called a degeneracy ordering of $G$, is as follows: for any vertex $v$ in $G$, the number of vertices that are larger than $v$ and adjacent to $v$ is at most $k$. We show an example of a degeneracy ordering of a graph in Fig. 1. Matula and Beck show that the degeneracy and a degeneracy ordering of $G$ can be obtained in $O(n + m)$ time [24]. Our proposed algorithm
EDS-D, shown in Algorithm 2, achieves amortized $O(k)$ time enumeration by using this good ordering. In what follows, we fix some degeneracy ordering of $G$ and number the indices of vertices from 1 to $n$ according to the degeneracy ordering. We assume that for each vertex $v$ and each dominating set $X$ of $G$, $N[v]$ is stored in a doubly linked list and sorted by the order. Note that the larger neighbors of $v$ can be listed in $O(k)$ time. Let us denote by $V^{<v} = \{1, 2, \ldots, v - 1\}$ and $V^{<v} = \{v + 1, \ldots, n\}$. Moreover, $A^{<v} = A \cap V^{<v}$ and $A^{<v} = A \cap V^{<v}$ for a subset $A$ of $V$. We first show the relation between $C(X)$ and $C(Y)$.

**Lemma 5.** Let $X$ be a dominating set of $G$ and $Y$ be a child of $X$. Then, $C(Y) \subseteq C(X)$.

From the Lemma 5, for any $v \in C(X)$, what we need to obtain the candidate set of $Y$ is to compute $\text{Del}(X, pv(Y)) = C(X) \setminus C(Y)$, where $Y = X \setminus \{v\}$. In addition, we can easily sort $C(Y)$ by the degeneracy ordering if $C(X)$ is sorted. In what follows, we denote by $\text{Del}_1(X, v) = \{u \in C(X)^{<v} \mid N[u] \cap X = \{u, v\}\}$, $\text{Del}_2(X, v) = \{u \in C(X)^{<v} \mid \exists w \in V \setminus (X \setminus \{v\})(N[w] \cap X = \{u, v\}\}$, and $\text{Del}_3(X, v) = C(X)^{\leq v}$. Next, we show the time complexity for obtaining $\text{Del}(X, pv(Y))$.

**Lemma 6.** For each $v \in C(X)$, $\text{Del}(X, v) = \text{Del}_1(X, v) \cup \text{Del}_2(X, v) \cup \text{Del}_3(X, v)$ holds.

We show an example of dominated list and a maintenance of $C(X)$ in Fig. 2. To compute a candidate set efficiently, for each vertex $u$ in $V$, we maintain the vertex lists $D_u(X)$ for $X$. We call $D_u(X)$ the dominated list of $u$ for $X$. The definition of $D_u(X)$ is as follows: If $u \in V \setminus X$, then $D_u(X) = N(u) \cap (X \setminus C(X))$. If $u \in X$, then $D_u(X) = N(u)^{<u} \cap (X \setminus C(X))$. For brevity, we write $D_u$ as $D_u(X)$ if no confusion arises. We denote by $D(X) = \bigcup_{u \in V} \{D_u\}$. By using $D(X)$, we can efficiently find $\text{Del}_1(X, v)$ and $\text{Del}_2(X, v)$.

**Lemma 7.** Let $X$ be a dominating set of $G$. Suppose that for each vertex $u$ in $G$, we can obtain the size of $D_u$ in constant time. Then, for each vertex $v \in C(X)$, we can compute $\text{Del}_1(X, v)$ in $O(k)$ time on average over all children of $X$.

**Lemma 8.** Suppose that for each vertex $w$ in $G$, we can obtain the size of $D_w$ in constant time. For each vertex $v \in C(X)$, we can compute $\text{Del}_2(X, v)$ in $O(k)$ time on average over all children of $X$.

In Lemma 7 and Lemma 8, we assume that the dominated lists were computed when we compute $\text{Del}(X, v)$ for each vertex $v$ in $C(X)$. We next consider how we maintain $D$. Next lemmas show the transformation from $D_u(X)$ to $D_u(Y)$ for each vertex $u$ in $G$. 
Lemma 9. Let $X$ be a dominating set, $v$ be a vertex in $C(X)$, and $Y = X \setminus \{v\}$. For each vertex $u \in G$ such that $u \neq v$, $D_u(Y) = D_u(X) \cup (N(u)^{<w} \cap (Del_1(X, v) \cup Del_2(X, v))) \cup (N(u)^{<w} \cap (Del_2(X, v) \setminus \{v\}))$.

Lemma 10. Let $X$ be a dominating set, $v$ be a vertex in $C(X)$, and $Y = X \setminus \{v\}$. $D_v(Y) = D_v(X) \cup (N(v)^{<w} \cap (Del_1(X, v) \cup Del_2(X, v))) \cup (N(v)^{<w} \cap X)$.

We next consider the time complexity for obtaining the dominated lists for children of $X$. From Lemma 9 and Lemma 10, a naive method for the computation needs $O(k |\text{Del}(X, v)| + k)$ time for each vertex $v$ of $X$ can list all larger neighbors of any vertex in $O(k)$ time. However, if we already know $C(W)$ and $D(W)$ for a child $W$ of $X$, then we can easily obtain $D(Y)$, where $Y$ is the child of $X$ immediately after $W$. The next lemma plays a key role in $\text{EDS-D}$. Here, for any two sets $A, B$, we denote by $A \oplus B = (A \setminus B) \cup (B \setminus A)$.

Lemma 11. Let $X$ be a dominating set, $v, u$ be vertices in $C(X)$ such that $u$ has the maximum index in $C(X)^{<w}$, $Y = X \setminus \{u\}$, and $W = X \setminus \{v\}$. Suppose that we already know $C(Y) \oplus C(W)$, $D(W)$, $\text{Del}(X, v)$, and $\text{Del}(X, u)$. Then, we can compute $D(Y)$ in $O(k |C(Y) \oplus C(W)| + k)$ time.

Proof. Suppose that $z$ is a vertex in $G$ such that $z \neq v$ and $z \neq u$. From the definition, $D_z(W) \setminus D_z(Y) = (\text{Del}(X, v) \setminus \text{Del}(X, u)) \cap N(z)^{<w}$ and $D_z(Y) \setminus D_z(W) = (\text{Del}(X, u) \setminus \text{Del}(X, v)) \cap N(z)^{<w}$. Hence, we first compute $\text{Del}(X, v) \oplus \text{Del}(X, u)$. Now, $(C(X) \setminus C(W)) \oplus (C(X) \setminus C(Y)) = C(W) \oplus C(Y)$. Next, for each vertex $c$ in $C(W) \oplus C(Y)$, we check whether we add to or remove $c$ from $D_z(Y)$ or not. Note that added or removed vertices from $D_z(Y)$ is a smaller neighbor of $z$. From the definition, if $c \notin D_z(Y)$ or $c \in D_z(X)$, then we add $c$ to $D_z(Y)$. Otherwise, we remove $c$ from $D_z(Y)$. Thus, since each vertex in $C(W) \oplus C(Y)$ has at most $k$ larger neighbors, for all vertices other than $u$ and $v$, we can compute the all dominated lists in $O(k |C(W) \oplus C(Y)|)$ time. Next we consider the update for $D_u(Y)$ and $D_v(Y)$. Note that from the definition, $D_u(W)$ and $D_v(Y)$ contain larger neighbors of $v$ and $u$, respectively. However, the number of such neighbors is $O(k)$. Finally, since $v$ belongs to $Y$, $v \in D_v(Z)$ if $u' \in N(v)^{<w}$ for any vertex $u'$. Thus, as with the above discussion, we can compute $D_u(Y)$ and $D_v(Y)$ in $O(k |C(W) \oplus C(Y)| + k)$ time.

Lemma 12. Let $X$ be a dominating set. Then, AllChildren$(X, C(X), D(X))$ of $\text{EDS-D}$ other than recursive calls can be done in $O(k |\text{ch}(X)| + k |\text{gch}(X)|)$ time.

Proof. We first consider the time complexity of $\text{Cand-D}$. From Lemma 7 and Lemma 8, $\text{Cand-D}$ correctly computes $\text{Del}_1(X, v)$ and $\text{Del}_2(X, v)$ in from line 18 to line 19 and from line 20 to line 21, respectively. For each loop from line 7, the algorithm picks the largest vertex in $C$. This can be done in $O(1)$ since $C$ is sorted. The algorithm needs to remove vertices in $\text{Del}_3(X, v)$. This can be done in line 9 and in $O(1)$ time since $v$ is the largest vertex. Thus, for each vertex $v$ in $C(X)$, $C(X \setminus \{v\})$ can be obtained in $O(k)$ time on average. Hence, for all vertices in $C(X)$, the candidate sets can be computed in $O(k |\text{ch}(X)|)$ time. Next, we consider the time complexity of $\text{DomList}$. Before computing $\text{DomList}$, $\text{EDS-D}$ already computed $C(Y) \oplus C(W)$, $\text{D}(W)$, $\text{Del}(X, v)$, and $\text{Del}(X, v')$. Note that we can compute $C(Y) \oplus C(W)$ when we compute $C(Y)$ and $C(W)$. Here, $W$ is the previous dominating set, $C'$ stores $C(W)$, and $D'$ stores $\text{D}(W)$. Thus, by using Lemma 11, we can compute $\text{D}(Y)$ in $O(k (|C(Y) \oplus C(W)| + k)$ time. In addition, for all vertices in $C(X)$, the dominated lists can be computed in $O(k |C(X)| + k |\text{gch}(X)|)$ time since $Y$ has at least $|C(W) \setminus C(Y)| - 1$ children and $|\text{gch}(X)|$ is at least the sum of $|C(W) \setminus C(Y)| - 1$ over all $Y \in \{X \setminus \{v\} \mid v \in C(X)\}$ and the previous solution $W$ of $Y$. When $\text{EDS-D}$ copies data
such as $D$, $EDS-D$ only copies the pointer of these data. By recording operations of each line, $EDS-D$ restores these data when backtracking happens. These restoring can be done in the same time of the above update computation.

\begin{theorem}
$EDS-D$ enumerates all dominating sets in $O(k)$ time per solution in a $k$-degenerate graph by using $O(n + m)$ space.
\end{theorem}

\textbf{Proof.} The parent-child relation of $EDS-D$ and $EDS$ are same. From Lemma 5 and Lemma 6, $EDS-D$ correctly computes all children. Hence, the correctness of $EDS-D$ is shown by the same manner of Theorem 4. We next consider the space complexity of $EDS-D$. For any vertex $v$ in $G$, if $v$ is removed from a data structure used in $EDS-D$ on a recursive procedure, $v$ will never be added to the data structure on descendant recursive procedures. In addition, for each recursive procedure, the number of data structures that are used in the procedure is constant. Hence, the space complexity of $EDS-D$ is $O(n + m)$. We finally consider the time complexity. Each recursive procedure needs $O(k |Cch(X)| + k |gch(X)|)$ time from Lemma 12. Thus, the time complexity of $EDS-D$ is $O(k \sum_{X \in S}(|Cch(X)| + |gch(X)|))$, where $S$ is the set of solutions. Now, $O(\sum_{X \in S}(|Cch(X)| + |gch(X)|)) = O(|S|)$. Hence, the statement holds.

\section{Efficient Enumeration for Graphs with Girth at Least Nine}

In this section, we propose an optimum enumeration algorithm $EDS-G$ for graphs with girth at least nine, where the girth of a graph is the length of a shortest cycle in the graph. That is, the proposed algorithm runs in constant amortized time per solution for such graphs. The algorithm is shown in Algorithm 3. To achieve constant amortized time enumeration, we focus on the local structure $G_v(X)$ for $(X, v)$ of $G$ defined as follows: 
$G_v(X) = G[(V \setminus N[X \setminus C(X)^{\leq v}]) \cup C(X)^{\leq v}]$. Fig. 3 shows an example of $G_v(X)$. $G_v(X)$ is a subgraph of $G$ induced by vertices that (1) are dominated by vertices only in $C(X)^{\leq v}$ or (2) are in $C(X)^{\leq v}$. Intuitively speaking, we can efficiently enumerate solutions by using the local structure and ignoring vertices in $G \setminus G_v(X)$ since the number of solutions that are generated according to the structure is enough to reduce the amortized time complexity to constant. We denote by $G(X) = G[(V \setminus N[X \setminus C(X)]) \cup C(X)]$ the local structure for $(X, v_*)$ of $G$, where $v_*$ is the largest vertex in $G$.

We first consider the correctness of $EDS-G$. The parent-child relation between solutions used in $EDS-G$ is the same as in $EDS$. Suppose that $X$ and $Y$ are dominating sets such that $X$ is the parent of $Y$. Recall that, from Lemma 6, $C(X) \setminus C(Y) = Del(X, v)$, where $X = Y \cup \{v\}$. We denote by $f_v(u, X) = \text{True}$ if there exists a neighbor $w$ of $u$ such that $w \in X \setminus C(X)^{\leq v}$; Otherwise $f_v(u, X) = \text{False}$. Thus, $\text{Cand-G}$ correctly computes $Del_1(X, v)$ and $Del_2(X, v)$ from line 17 to 19. Moreover, in line 14, vertices in $Del_3(X, v)$ are removed from $C(X)$ and hence, $\text{Cand-G}$ also correctly computes $C(X \setminus \{v\})$. Moreover, for each vertex $w$ removed from $G$ during enumeration, $w$ is dominated by some vertices in $G$. Hence, by the same discussion as Theorem 4, we can show that $EDS-G$ enumerates all dominating sets. In the remaining of this section, we show the time complexity of $EDS-G$. Note that $G_v(X)$ does not include any vertex in $N[Del_3(X, v)] \setminus C(X)^{\leq v}$. Hence, we will consider only vertices in $Del_1(X, v) \cup Del_2(X, v) \cup \{v\}$. We denote by $Del'(X, v) = Del_1(X, v) \cup Del_2(X, v) \cup \{v\}$. We first show the time complexity for updating the candidate sets.

In what follows, if $v$ is the largest vertex in $C(X)$, then we simply write $f(u, X)$ as $f_v(u, X)$. We denote by $N'_v(u) = N_{G_v(X)}(u)$, $N''_v[u] = N'_v(u) \cup \{u\}$, and $d''_v(u) = |N''_v[u]|$ if no confusion arises. Suppose that $G$ and $G_v(X)$ are stored in an adjacency list, and neighbors of a vertex are stored in a doubly linked list and sorted in the ordering.
Algorithm 3: EDS-G enumerates all dominating sets in $O(1)$ time per solution for a graph with girth at least nine.

1. **Procedure** EDS-G($G = (V,E)$) // $G$: an input graph
   2. for $v \in V$ do $f_v \leftarrow$ False;
   3. AllChildren($V,V,\{f_1, \ldots, f_{|V|}\}, G$);
   4. **Procedure** AllChildren($X,C,F,G$)
      5. Output $X$;
      6. for $v \in C'(X)$ do // $v$ is the largest vertex in $C$
         7. $Y \leftarrow X \setminus \{v\}$;
         8. $(C(Y), F(Y), G(Y)) \leftarrow$ Cand-G($v, C, F, G$);
         9. AllChildren($Y, C(Y), F(Y), G(Y)$);
         10. for $u \in N_G(v)$ do
             11. if $u \in C$ then $f_u \leftarrow$ True;
             12. else $G \leftarrow G \setminus \{u\}$;
             13. $G \leftarrow G \setminus \{v\}$;
             14. $C \leftarrow C \setminus \{v\}$; // Remove vertices in Del$_3(X,v)$.
   15. **Procedure** Cand-G($v, C, F, G$)
      16. $\text{Del}_1 \leftarrow \emptyset$; $\text{Del}_2 \leftarrow \emptyset$;
      17. for $u \in N_G(v)$ do
         18. if $N_G[u] \cap X = \{u,v\}$ and $f_u = \text{False}$ then $\text{Del}_1 \leftarrow \text{Del}_1 \cup \{u\}$;
      19. else if $\exists w(N_G[u] \cap X = \{w,v\})$ then $\text{Del}_2 \leftarrow \text{Del}_2 \cup \{w\}$;
      20. $C' \leftarrow C \setminus (\text{Del}_1 \cup \text{Del}_2 \cup \{v\})$;
      21. for $u \in N'[\text{Del}_1 \cup \text{Del}_2]$ do // Lemma 17
         22. $f_u \leftarrow$ True;
      23. if $u \notin C'$ then $G \leftarrow G \setminus \{u\}$;
      24. if $f_u = \text{True}$ then $G \leftarrow G \setminus \{v\}$;
      25. return $(C', F, G)$;

**Lemma 14.** Let $X$ be a dominating set, $v$ be a vertex in $C(X)$, and $u$ be a vertex in $G$. Then, $u \in \text{Del}_1(X,v)$ if and only if $N^*_v[u] \cap X = \{u,v\}$ and $f_u(u, X) = \text{False}$.

**Lemma 15.** Let $X$ be a dominating set, $v$ be a vertex in $C(X)$, and $u$ be a vertex in $G$. Then, $u \in \text{Del}_2(X,v)$ if and only if there is a vertex $w$ in $G_v(X)$ such that $N^*_v[w] \cap X = \{u,v\}$.

**Lemma 16.** Let $X$ be a dominating set and $v$ be a vertex in $C(X)$. Suppose that for any vertex $u$, we can check the number of $u$’s neighbors in the local structure $G_v(X)$ and the value of $f_u(u, X)$ in constant time. Then, we can compute $C(X \setminus \{v\})$ from $C(X)^{\leq v}$ in $O(d'_v(v))$ time.

**Lemma 17.** Let $X$ be a dominating set, $v$ be a vertex in $C(X)$, and $Y = X \setminus \{v\}$. Then, we can compute $G(Y)$ from $G_v(X)$ in $O \left( \sum_{u \in \text{Del}'(X,v)} d'_v(u) + \sum_{u \in G_v(X) \setminus G(Y)} d'_v(u) \right)$ time. Note that $N'_v(u) = N_{G_v(X)}(u)$ and $d'_v(u) = |N'_v(u)|$.

From Lemma 16 and Lemma 17, we can compute the local structure and the candidate set of $Y$ from those of $X$ in $O \left( \sum_{u \in \text{Del}'(X,v)} d'_v(u) + \sum_{u \in G_v(X) \setminus G(Y)} d'_v(u) \right)$ time. We next consider the time complexity of the loop in line 10. In this loop procedure, EDS-G deletes all the neighbors $u$ of $v$ from $G_v(X)$ if $u \notin C(X)^{\leq v}$ because for each descendant $W$ of dominating set $Y'$, $v \in W \setminus C(W)$, where $Y'$ is a child of $X$ and is generated after $Y$. Thus,
this needs $O \left( d'_v(v) + \sum_{u \in N'_v(v) \setminus X} d'_u(u) \right)$ time. Hence, from the above discussion, we can obtain the following lemma:

\[ \begin{align*}
&\textbf{Lemma 18.} \text{ Let } X \text{ be a dominating set, } v \text{ be a vertex in } C(X), \text{ and } Y = X \setminus \{v\}. \text{ Then, AllChildren other than a recursive call runs in the following time bound:} \\
&O \left( \sum_{u \in Del'(X,v)} d'_u(u) + \sum_{u \in G_u(X) \setminus G(Y)} d'_u(u) + \sum_{u \in N'_v(v) \setminus X} d'_u(u) \right). 
\end{align*} \]  

Before we analyze the number of descendants of $X$, we show the following lemmas.

\[ \begin{align*}
&\textbf{Lemma 19.} \text{ Let us denote by } Pen_v(X) = \{u \in Del'(X,v) \mid d'_u(u) = 1\}. \text{ Then, } \\
&\sum_{v \in C(X)} |Pen_v(X)| \text{ is at most } |C(X)|.
\end{align*} \]

Let $v$ be a vertex in $C(X)$ and a pendant in $G_v(X)$. Since the number of such pendants is at most $|C(X)|$, the sum of degree of such pendants is at most $|C(X)|$ in each execution of AllChildren without recursive calls. Hence, the cost of deleting such pendants is $O(|C(X)|)$ time. Next, we consider the number of descendants of $X$. From Lemma 19, we can ignore such pendant vertices. Hence, for each $u \in Del'(X,v)$, we will assume that $d'_u(u) \geq 2$ below.

\[ \begin{align*}
&\textbf{Lemma 20.} \text{ Let } X \text{ be a dominating set, } v \text{ be a vertex in } C(X), \text{ and } Y \text{ be a dominating set } X \setminus \{v\}. \text{ Then, } |C(Y)| \text{ is at least } |N'_v(v) \cap X| - Del'(X,v)|. 
\end{align*} \]

\[ \begin{align*}
&\textbf{Lemma 21.} \text{ Let } X \text{ be a dominating set, } v \text{ be a vertex in } C(X), \text{ and } Y \text{ be a dominating set } X \setminus \{v\}. \text{ Then, } |C(Y)| \text{ is at least } \sum_{u \in N'_v(v) \setminus X} (d'_u(u) - 1). 
\end{align*} \]

\[ \begin{align*}
&\textbf{Lemma 22.} \text{ Let } X \text{ be a dominating set, } v \text{ be a vertex in } C(X), \text{ and } Y \text{ be a dominating set } X \setminus \{v\}. \text{ Then, } |C(Y)| \text{ is at least } \sum_{u \in Del'(X,v) \setminus \{v\}} (d'_u(u) - 1). 
\end{align*} \]

\[ \begin{align*}
&\textbf{Lemma 23.} \text{ Let } X \text{ be a dominating set } v \text{ be a vertex in } C(X), \text{ and } Y \text{ be a dominating set } X \setminus \{v\}. \text{ Then, the number of children and grandchildren of } Y \text{ is at least } \sum_{u \in G_v(X) \setminus (G(Y) \cup Del'(X,v) \cup N'_v(v))} (d'_u(u) - 1). 
\end{align*} \]

Note that for any pair of candidate vertices $v$ and $v'$, $X \setminus \{v\}$ and $X \setminus \{v'\}$ do not share their descendants. Thus, from Lemma 20, Lemma 21, Lemma 22, and Lemma 23, we can obtain the following lemma:
Lemma 24. Let $X$ be a dominating set. Then, the sum of the number of $X$’s children, grandchildren, and great-grandchildren is bounded by the following order:

$$\Omega \left( |C(X)| + \sum_{v \in C(X)} \left( \sum_{u \in Del(v)(X,v)} d'_v(u) + \sum_{u \in G_v(X) \setminus G(Y)} d'_v(u) + \sum_{u \in N'_v(v) \setminus X} d'_v(u) \right) \right). \quad (2)$$

From Lemma 18, Lemma 19, and Lemma 24, each iteration outputs a solution in constant amortized time. Hence, by the same discussion of Theorem 13, we can obtain the following theorem.

Theorem 25. For an input graph with girth at least nine, $\text{EDS-G}$ enumerates all dominating sets in $O(1)$ time per solution by using $O(n + m)$ space.

Proof. The correctness of $\text{EDS-G}$ is shown by Theorem 4, Lemma 14, and Lemma 15. By the same discussion with Theorem 13, the space complexity of $\text{EDS-G}$ is $O(n + m)$. We next consider the time complexity of $\text{EDS-G}$. From Lemma 18, Lemma 19, and Lemma 24, we can amortize the cost of each recursion by distributing $O(1)$ time cost to the corresponding descendant discussed in the above lemmas. Thus, the amortized time complexity of each recursion becomes $O(1)$. Moreover, each recursion outputs a solution. Hence, $\text{EDS-G}$ enumerates all solutions in $O(1)$ amortized time per solution.

5 Conclusion

In this paper, we proposed two enumeration algorithms. $\text{EDS-D}$ solves the dominating set enumeration problem in $O(k)$ time per solution by using $O(n + m)$ space, where $k$ is a degeneracy of an input graph $G$. Moreover, $\text{EDS-G}$ solves this problem in constant time per solution if an input graph has girth at least nine.

Our future work includes to develop efficient dominating set enumeration algorithms for dense graphs. If a graph is dense, then $k$ is large and $G$ has many dominating sets. For example, in the case of complete graphs, $k$ is equal to $n - 1$ and every nonempty subset of $V$ is a dominating set. That is, the number of solutions for a dense graph is much larger than that for a sparse graph. This allows us to spend more time in each recursive call. However, $\text{EDS-D}$ is not efficient for dense graphs although the number of solutions is large. Moreover, if $G$ is small girth, that is, $G$ is dense then $\text{EDS-G}$ does not achieve constant amortized time enumeration. Hence, the dominating set enumeration problem for dense graphs is interesting.

References

Efficient Enumeration of Dominating Sets for Sparse Graphs


Abstract

The classic TQBF problem is to determine who has a winning strategy in a game played on a given CNF formula, where the two players alternate turns picking truth values for the variables in a given order, and the winner is determined by whether the CNF gets satisfied. We study variants of this game in which the variables may be played in any order, and each turn consists of picking a remaining variable and a truth value for it.

- For the version where the set of variables is partitioned into two halves and each player may only pick variables from his/her half, we prove that the problem is \( \text{PSPACE} \)-complete for 5-CNFs and in \( \text{P} \) for 2-CNFs. Previously, it was known to be \( \text{PSPACE} \)-complete for unbounded-width CNFs (Schaefer, STOC 1976).
- For the general unordered version (where each variable can be picked by either player), we also prove that the problem is \( \text{PSPACE} \)-complete for 5-CNFs and in \( \text{P} \) for 2-CNFs. Previously, it was known to be \( \text{PSPACE} \)-complete for 6-CNFs (Ahlroth and Orponen, MFCS 2012) and \( \text{PSPACE} \)-complete for positive 11-CNFs (Schaefer, STOC 1976).

1 Introduction

 Conjunctive normal form formulas (CNFs) are among the most prevalent representations of boolean functions. All sorts of computational problems concerning CNFs – such as satisfying them, minimizing them, refuting them, fooling them, and playing games on them – play central roles in complexity theory. A CNF is a conjunction of clauses, where each clause is a disjunction of literals; a \( w \)-CNF has at most \( w \) literals per clause. The width \( w \) is often the most important parameter governing the complexity of problems concerning CNFs. The following are three classical games played on a CNF \( \varphi(x_1, \ldots, x_n) \):

- In the ordered game, player 1 assigns a bit value for \( x_1 \), then player 2 assigns \( x_2 \), then player 1 assigns \( x_3 \), and so on, and the winner is determined by whether \( \varphi \) gets satisfied.

 Note that the variables must be played in the prescribed order \( x_1, x_2, x_3, \ldots \). Deciding
who has a winning strategy – better known as TQBF or QSAT – is \( \text{PSPACE} \)-complete for 3-CNFs [11] and in \( \text{P} \) for 2-CNFs [2, 6]. Many \( \text{PSPACE} \)-completeness results have been shown by reducing from the ordered 3-CNF game.

In the unordered game, each player is allowed to pick which remaining variable to play next (as well as which bit value to assign it), and again the winner is determined by whether \( \varphi \) gets satisfied. Deciding who has a winning strategy is \( \text{PSPACE} \)-complete for 6-CNFs [1] and for 11-CNFs with only positive literals [9, 10]. The unordered game on positive CNFs is also known as the maker–breaker game, and a simplified proof of \( \text{PSPACE} \)-completeness for unbounded-width positive CNFs appears in [5]. Many \( \text{PSPACE} \)-completeness results have been proven by reducing from the unordered positive CNF game [7, 5, 8]. For the general unordered CNF game, nothing was known for width \( \leq 6 \); in particular, the complexity of the unordered 2-CNF game was not studied in the literature before.

In the partitioned game, the set of variables is partitioned into two halves and each player may only pick variables from his/her half. This is, in a sense, intermediate between ordered and unordered: the ordered game restricts the set of variables available to each player and the order they must be played; the unordered game restricts neither; the partitioned game restricts only the former. Deciding who has a winning strategy was shown to be \( \text{PSPACE} \)-complete for unbounded-width CNFs in [9, 10], where it was explicitly posed as an open problem to show \( \text{PSPACE} \)-completeness with any constant bound on the width. This game has been used for \( \text{PSPACE} \)-completeness reductions [3], and a variant with a matching between the two players’ variables has also been studied [4]. The partitioned 2-CNF game was not studied in the literature before.

We prove that the unordered and partitioned games are both \( \text{PSPACE} \)-complete for 5-CNFs; the former improves the width 6 bound from [1], and the latter resolves the 42-year-old open problem from [9, 10]. We also prove that the unordered and partitioned games are both in \( \text{P} \) for 2-CNFs. The complexity for width 3 and 4 remains open. In the following section we give the precise definitions and theorem statements.

### 1.1 Statement of results

The unordered CNF game is defined as follows. There are two players, denoted T (for “true”) and F (for “false”). The input consists of a CNF \( \varphi \), a set of variables \( X = \{x_1, \ldots, x_n\} \) containing all the variables that appear in \( \varphi \) (and possibly more), and a specification of which player goes first. The players alternate turns, and each turn consists of picking a remaining variable from \( X \) and assigning it a value 0 or 1. Once all variables have been assigned, the game ends and T wins if \( \varphi \) is satisfied, and F wins if it is not. We let \( G \) (for “game”) denote the problem of deciding which player has a winning strategy, given \( \varphi \), \( X \), and who goes first.

The partitioned CNF game is similar to the unordered CNF game, except that \( X \) is partitioned into two halves \( X_T \) and \( X_F \), and each player may only pick variables from his/her half. If \( n \) is even we require \( |X_T| = |X_F| \), and if \( n \) is odd we require \( |X_T| = |X_F| + 1 \) if T goes first, and \( |X_F| = |X_T| + 1 \) if F goes first. We let \( G^\% \) denote the problem of deciding which player has a winning strategy, given \( \varphi \), the partition \( X = X_T \cup X_F \), and who goes first.

We let \( G_w \) and \( G_w^\% \) denote the restrictions of \( G \) and \( G^\% \), respectively, to instances where \( \varphi \) has width \( w \), i.e., each clause has at most \( w \) literals. Now, we state our results as the following theorems:

▶ **Theorem 1.** \( G_5 \) is \( \text{PSPACE} \)-complete.
Theorem 2. \( G^5 \) is \( \text{PSPACE} \)-complete.

Theorem 3. \( G_2 \) is in \( \text{P} \), in fact, in \( \text{Linear Time} \).

Theorem 4. \( G^{\%} \) is in \( \text{P} \), in fact, in \( \text{Linear Time} \).

We prove Theorem 1 and Theorem 2 in Section 2 by showing reductions from the \( \text{PSPACE} \)-complete games \( G \) and \( G^5 \) respectively. For Theorem 3 and Theorem 4 in Section 3 we prove characterizations in terms of the graph representation from the classical \( 2\)-SAT algorithm – who has a winning strategy in terms of certain graph properties – and we design linear time algorithms to check these properties.\(^1\)

In the proofs, it is helpful to distinguish four patterns for “who goes first” and “who goes last”, we introduce new subscripts. For \( a, b \in \{T, F\} \), the subscript \( a \_\_\_ b \_\_\_ \) means player \( a \) goes first and player \( b \) goes last, \( a \_\_\_ \) means \( a \) goes first, and \( \_\_\_ b \) means \( b \) goes last. These may be combined with the width \( w \) subscript. For example, \( G^\%_{T\_\_\_ F} \) (which was denoted \( G^\%_{\text{free}}(\text{CNF}) \) in \([9, 10]\), by the way) corresponds to the partitioned game where \( T \) goes first and \( F \) goes last (so \( n \) must be even), and \( G^5_{T\_\_\_} \) corresponds to the unordered game with width 5 where \( T \) goes last (so either \( n \) is even and \( F \) goes first, or \( n \) is odd and \( T \) goes first).

2 5-CNF

We prove Theorem 1 in Section 2.1 and Theorem 2 in Section 2.2.

2.1 \( G_5 \)

In this section we prove Theorem 1. It is trivial to argue that \( G_5 \in \text{PSPACE} \). We prove \( \text{PSPACE} \)-hardness by showing a reduction \( G^\%_{T\_\_\_ F} \leq G_5_{T\_\_\_ F} \) in Section 2.1.2. \( G^\%_{T\_\_\_ F} \) is already known to be \( \text{PSPACE} \)-complete \([9, 10, 5, 1]\). We will talk about the other three patterns \( G^\%_{F\_\_\_ F}, G^\%_{T\_\_\_ T}, G^\%_{F\_\_\_ T} \) in the full version. Before the formal proof we develop the intuition in Section 2.1.1.

2.1.1 Intuition

In \( \text{NP} \)-completeness, recall the following simple reduction from SAT with unbounded width to 3-SAT. Suppose a SAT instance is given by \( \varphi \) over set of variables \( X \). If \((\ell_1 \lor \ell_2 \lor \ell_3 \lor \cdots \lor \ell_k)\) is a clause in \( \varphi \) with width \( k > 3 \), then the reduction introduces fresh variables \( z_1, z_2, \ldots, z_{k-1} \) and generates a chain of clauses in \( \varphi' \) as follows:

\[
(\ell_1 \lor z_1) \land (\overline{z}_1 \lor \ell_2 \lor z_2) \land \cdots \land (\overline{z}_{k-1} \lor \ell_k \lor z_k) \land (\overline{z}_{k-1} \lor \ell_k)
\]

Each clause of \( \varphi \) gets a separate set of fresh variables for its chain, and we let \( Z = \{z_1, z_2, \ldots\} \) be the set of all fresh variables for all chains. The reduction claims that \( \varphi \) is satisfiable if and only if \( \varphi' \) is satisfiable. We are going to have a stronger property in Claim 1.

Claim 1. For every assignment \( x \) to \( X \): \( \varphi(x) \) is satisfied iff there exists an assignment \( z \) to \( Z \) such that \( \varphi'(x, z) \) is satisfied.

\(^1\) We remark that it is not automatic that two-player games on 2-CNFs are solvable in polynomial time; e.g., the game played on a 2-CNF with only negative literals in which players alternate turns assigning variables of their choice to 0 and where the loser is the first to falsify the 2-CNF, as well as the partitioned variant of this game, are \( \text{PSPACE} \)-complete \([9, 10]\).
Proof. Suppose \( x \) satisfies \( \varphi \). If \( x \) satisfies \((\ell_1 \lor \ell_2 \lor \ell_3 \lor \cdots \lor \ell_k)\) in \( \varphi \) by \( \ell_i = 1 \), then in the corresponding chain of clauses in \( \varphi' \), the clause having \( \ell_i \) also gets satisfied by \( \ell_i = 1 \) and the rest of the clauses in that chain can get satisfied by assigning all \( z \)’s on the left side of \( \ell_i \) as 1 and right side of \( \ell_i \) as 0.

Now suppose \( x \) does not satisfy \( \varphi \). Then at least one of the clauses of \( \varphi \) has all literals assigned as 0. The corresponding chain of clauses in \( \varphi' \) essentially becomes:

\[
(z_1) \land (z_1 \lor z_2) \land \cdots \land (z_{i-1} \lor z_i) \land \cdots \land (z_{k-2} \lor z_{k-1}) \land (z_{k-1})
\]

In order to satisfy the above chain, \( z_1 = 1 \) and \( z_{k-1} = 0 \). It also introduces the following chain of implications: \( z_1 \Rightarrow z_2 \Rightarrow z_3 \Rightarrow \cdots \Rightarrow z_{k-1} \). Following the chain we get \( (z_1 \Rightarrow z_{k-1}) = (1 \Rightarrow 0) \). Therefore, we conclude that \( \varphi'(x, z) \) cannot be satisfied for any assignment \( z \). ▶

Now this reduction does not show \( G_{T \ldots F} \leq G_{3,T \ldots F} \) since the games on \( \varphi \) and \( \varphi' \) are not equivalent. We show a simple example to make our point. Consider the following \( G_{T \ldots F} \) game over variables \( \{x_0, x_1, \ldots, x_k\} \).

\[
\varphi = x_0 \land (x_1 \lor x_2 \lor x_3 \lor \cdots \lor x_k), \text{ where } k > 1
\]

In the above \( G_{T \ldots F} \) game, \( T \) has a winning strategy: On the first move \( T \) plays \( x_0 = 1 \). Then whatever \( F \) plays, \( T \) plays one of the \( k-1 \) many unassigned \( x_i \) from \( \{x_1, x_2, \ldots, x_k\} \) as 1. \( T \) wins.

But if we introduce fresh variables \( \{z_1, z_2, z_3, \ldots\} \) as in the NP-completeness reduction then we get a game over variables \( \{x_0, x_1, x_2, \ldots, x_k\} \cup \{z_1, z_2, \ldots, z_{k-1}\} \):

\[
\varphi' = x_0 \land (x_1 \lor z_1) \land \cdots \land (z_{i-1} \lor x_i \lor z_i) \land \cdots \land (z_{k-1} \lor x_k)
\]

In the above \( G_{3,T \ldots F} \) game, \( F \) has a winning strategy: On the first move \( T \) must play \( x_0 = 1 \), otherwise \( F \) wins by \( x_0 = 0 \). Then \( F \) plays \( x_1 = 0 \) and \( T \) must reply by \( z_1 = 1 \), otherwise \( F \) wins by \( z_1 = 0 \). Then \( F \) plays \( x_2 = 0 \) and \( T \) must reply by \( z_2 = 1 \), otherwise \( F \) wins by \( z_2 = 0 \). The strategy goes on like this until the last clause and \( F \) wins by \( x_k = 0 \).

The \( G_{3,T \ldots F} \) game is disadvantageous for \( T \) compared to the \( G_{T \ldots F} \) game. The disadvantage arises from \( F \) having the beginning move in a fresh chain of clauses.

Now the intuition is to design a game version of the NP-completeness reduction by fixing the imbalance. We design \( \psi \) in such a way that the games on \( \varphi \) and \( \psi \) stay equivalent. In order to counter the unfairness for \( T \) due to fresh variables \( \{z_1, z_2, z_3, \ldots\} \), we replace \( z_i \) by a pair of variables \((a_i, b_i)\) which gives \( T \) more opportunities to satisfy the clauses. The construction of a chain of clauses in \( \psi \) from a clause \((\ell_1 \lor \ell_2 \lor \ell_3 \lor \cdots \lor \ell_k)\) in \( \varphi \) goes as follows:

\[
(\ell_1 \lor a_1 \lor b_1) \land \cdots \land (\overline{a}_{i-1} \lor \overline{b}_{i-1} \lor \ell_i \lor a_i \lor b_i) \land \cdots \land (\overline{a}_{k-1} \lor \overline{b}_{k-1} \lor \ell_k)
\]

We just constructed a 5-CNF \( \psi \). Let us consider the \( G_{\alpha,T \ldots F} \) game on \( \psi \). In an optimal gameplay, no player should play \( a \)'s or \( b \)'s before playing \( x \)'s. Intuitively, this is because, if \( F \) plays any \( a_i \) or \( b_i \), then \( T \) can reply by making \( a_i \neq b_i \) and both clauses involving \( a_i \) and \( b_i \) will be satisfied, which benefits \( T \). If \( T \) plays any \( a_i \) or \( b_i \), \( F \) can reply by making \( a_i = b_i \), which satisfies one clause involving \( a_i \) and \( b_i \) but the other clause gets two 0 literals. Since only one of the two clauses gets satisfied by \( a_i, b_i \), \( T \) would like to wait for more information before deciding which one to satisfy with \( a_i, b_i \): it depends on whether they are on the right side or left side of a satisfied \( \ell_i \) in a chain, which in turn depends on the assignment \( x \).
So, an optimal gameplay consists of two phases. In the first phase, players should play only \( x \)'s. Whoever deviates from this optimal strategy does not have the upper hand. The second phase begins when all the \( x \)'s have been played and someone must start playing \( a \)'s and \( b \)'s. Since the number of fresh variables is even (\( 2|Z| \)) and \( F \) plays last, \( T \) must be the one to start the second phase, which is essential since if \( F \) started the second phase then \( T \) could satisfy all the clauses regardless of what happened in the first phase. This observation also allows us to show PSPACE-completeness of \( G_{5,F,T} \), discussed in the full version.

In the second phase, after \( T \) plays any \( a_i \) or \( b_i \), it is optimal for \( F \) to reply by making \( a_i = b_i \). Assuming this optimal gameplay by \( F \), we can consider a pair \( (a_i,b_i) \) as a single variable \( z_i \) which can be assigned only by \( T \). Effectively, the second phase just consists of \( T \) choosing an assignment \( z \) to \( \varphi' \) from the NP-completeness reduction. Thus \( \psi(x,a,b) \) is satisfied iff \( \varphi'(x,z) \) is satisfied, which by Claim 1 is possible iff \( \varphi(x) \) is satisfied, where \( x \) is the assignment from the first phase.

### 2.1.2 Formal Proof

We show \( G_{T,F} \subseteq G_{5,T,F} \). Suppose an instance of \( G_{T,F} \) is given by \( (\varphi,X) \) where \( \varphi \) is a CNF with unbounded width over set of variables \( X \). We show how to construct an instance \( (\psi,Y) \) for \( G_{5,T,F} \) where \( \psi \) is a 5-CNF over set of variables \( Y \). Suppose \( (\ell_1 \vee \ell_2 \vee \ell_3 \vee \cdots \vee \ell_k) \) is a clause in \( \varphi \). If \( k \leq 3 \), the same clause remains in \( \psi \). If \( k > 3 \), we show how to construct a chain of clauses in \( \psi \). We introduce two sets of fresh variables \( \{a_1,a_2,a_3,\ldots,a_{k-1}\} \) and \( \{b_1,b_2,b_3,\ldots,b_{k-1}\} \) as follows:

\[
(\ell_1 \lor a_1 \lor b_1) \land \cdots \land (\overline{a_{i-1}} \lor \overline{b_{i-1}} \lor \ell_i \lor a_i \lor b_i) \land \cdots \land (\overline{a_{k-1}} \lor \overline{b_{k-1}} \lor \ell_k)
\]

Each clause of \( \varphi \) gets separate sets of fresh variables for its chain, and we let \( A = \{a_1,a_2,a_3,\ldots\} \) and \( B = \{b_1,b_2,b_3,\ldots\} \) be the sets of all fresh variables for all chains. Finally we get a 5-CNF \( \psi \) over set of variables \( Y = X \cup A \cup B \).

We claim that \( T \) has a winning strategy in \( (\varphi,X) \) iff \( T \) has a winning strategy in \( (\psi,Y) \).

Suppose \( T \) has a winning strategy in \( (\varphi,X) \). We describe \( T \)'s winning strategy in \( (\psi,Y) \) as Algorithm 1. To see that the strategy works, note that the winning strategy in \( (\varphi,X) \) ensures that \( \varphi(x) \) is satisfied by the assignment \( x \) to \( X \) in the first phase, so according to 1, there is an assignment \( z \) to \( Z \) such that \( \varphi'(x,z) \) is satisfied. \( T \) can ensure that for each \( i \), either \( a_i = z_i \) or \( b_i = z_i \) (since \( a_i = z_i \) or \( b_i = z_i \) due to line 8, or \( a_i \neq b_i \) due to line 4 or line 7) and thus \( \psi(x,a,b) \) gets satisfied, since \( \varphi'(x,z) \) is satisfied and each clause of \( \psi \) is identical to a clause from \( \varphi' \) but with each \( z_i \) replaced with \( a_i \lor b_i \) and \( \overline{z_i} \) replaced with \( \overline{a_i} \lor \overline{b_i} \).

Suppose \( F \) has a winning strategy in \( (\varphi,X) \). We describe \( F \)'s winning strategy in \( (\psi,Y) \) as Algorithm 2. To see that the strategy works, note that the winning strategy in \( (\varphi,X) \) ensures that \( \varphi(x) \) is unsatisfied by the assignment \( x \) to \( X \), so according to Claim 1, for all assignments \( z \) to \( Z \), \( \varphi'(x,z) \) is unsatisfied. \( F \) can ensure that for each \( i \), \( a_i = b_i \); let us call this common value \( z_i \). Thus \( \psi(x,a,b) \) is unsatisfied, since \( \varphi'(x,z) \) is unsatisfied and \( \psi(x,a,b) = \varphi'(x,z) \).

### 2.2 \( G_{5}^5 \)

In this section we prove Theorem 2. It is trivial to argue that \( G_{5}^5 \in \text{PSPACE} \). We prove PSPACE-hardness by showing a reduction \( G_{5,F,T}^5 \leq G_{5,F,T}^5 \) in Section 2.2.2. \( G_{T,F}^5 \) is already known to be PSPACE-complete [9, 10]. We will talk about the other three patterns \( G_{F,T}^5 \), \( G_{F,T}^5 \), \( G_{F,T}^5 \) in the full version. Before the formal proof we develop the intuition in Section 2.2.1.
We show $G$ is 

This intuition is a continuation of Section 2.1. The reduction is the same as $G \leq G_5$. Suppose an instance of $G \leq G_5$ is given by $(\varphi, X_T, X_F)$ where $\varphi$ is a CNF with unbounded width over sets of variables $X_T$ and $X_F$. We show how to construct an instance $(\psi, Y_T, Y_F)$ for $G \leq G_5$ where $\psi$ is a 5-CNF over sets of variables $Y_T$ and $Y_F$. Suppose $(\ell_1 \lor \ell_2 \lor \ell_3 \lor \cdots \lor \ell_k)$ is a clause in $\varphi$. If $k \leq 3$, the same clause remains in $\psi$. If $k > 3$, we show how to construct a chain of clauses in $\psi$. We introduce two sets of fresh variables $\{a_1, a_2, a_3, \ldots, a_{k-1}\}$ for $T$ and $\{b_1, b_2, b_3, \ldots, b_{k-1}\}$ for $F$ as follows:

Each clause of $\varphi$ gets separate sets of fresh variables for its chain, and we let $A = \{a_1, a_2, a_3, \ldots\}$ for $T$ and $B = \{b_1, b_2, b_3, \ldots\}$ for $F$ be the sets of all fresh variables for all chains. Finally we get a 5-CNF $\psi$ over sets of variables $Y_T = X_T \cup A$ and $Y_F = X_F \cup B$. We claim that $T$ has a winning strategy in $(\varphi, X_T, X_F)$ if $T$ has a winning strategy in $(\psi, Y_T, Y_F)$.

### Algorithm 1: $T$'s winning strategy in $(\psi, Y)$ when $T$ has a winning strategy in $(\varphi, X)$.

1. while there is a remaining $X$-variable do
2. if (first move) or (F played an $X$-variable in the previous move) then
3. play according to the same winning strategy as in $(\varphi, X)$
4. else if $F$ played $a_i$ or $b_i$ in the previous move then play the other one to make $a_i \neq b_i$
5. while there is a remaining $A$-variable or $B$-variable do
6. if (F played $a_i$ or $b_i$ in the previous move) and (one of $a_i$ or $b_i$ remains unplayed) then
7. play the other one to make $a_i \neq b_i$
8. else pick a remaining $a_i$ or $b_i$ and assign it $z_i$'s value from Claim 1

### Algorithm 2: $F$'s winning strategy in $(\psi, Y)$ when $F$ has a winning strategy in $(\varphi, X)$.

1. while there is a remaining variable do
2. if $T$ played an $X$-variable in the previous move then
3. play according to the same winning strategy as in $(\varphi, X)$
4. else if $T$ played $a_i$ or $b_i$ in the previous move then play the other one to make $a_i = b_i$

#### 2.2.1 Intuition

This intuition is a continuation of Section 2.1.1. The reduction is the same as $G_T \leq G_5$. Suppose an instance of $G_T \leq G_5$ is given by $(\varphi, X_T, X_F)$ where $\varphi$ is a CNF with unbounded width over sets of variables $X_T$ and $X_F$. We show how to construct an instance $(\psi, Y_T, Y_F)$ for $G_T \leq G_5$ where $\psi$ is a 5-CNF over sets of variables $Y_T$ and $Y_F$. Suppose $(\ell_1 \lor \ell_2 \lor \ell_3 \lor \cdots \lor \ell_k)$ is a clause in $\varphi$. If $k \leq 3$, the same clause remains in $\psi$. If $k > 3$, we show how to construct a chain of clauses in $\psi$. We introduce two sets of fresh variables $\{a_1, a_2, a_3, \ldots, a_{k-1}\}$ for $T$ and $\{b_1, b_2, b_3, \ldots, b_{k-1}\}$ for $F$ as follows:

Each clause of $\varphi$ gets separate sets of fresh variables for its chain, and we let $A = \{a_1, a_2, a_3, \ldots\}$ for $T$ and $B = \{b_1, b_2, b_3, \ldots\}$ for $F$ be the sets of all fresh variables for all chains. Finally we get a 5-CNF $\psi$ over sets of variables $Y_T = X_T \cup A$ and $Y_F = X_F \cup B$. We claim that $T$ has a winning strategy in $(\varphi, X_T, X_F)$ if $T$ has a winning strategy in $(\psi, Y_T, Y_F)$. 

### Formal Proof

We show $G_T \leq G_5$. Suppose an instance of $G_T \leq G_5$ is given by $(\varphi, X_T, X_F)$ where $\varphi$ is a CNF with unbounded width over sets of variables $X_T$ and $X_F$. We show how to construct an instance $(\psi, Y_T, Y_F)$ for $G_T \leq G_5$ where $\psi$ is a 5-CNF over sets of variables $Y_T$ and $Y_F$. Suppose $(\ell_1 \lor \ell_2 \lor \ell_3 \lor \cdots \lor \ell_k)$ is a clause in $\varphi$. If $k \leq 3$, the same clause remains in $\psi$. If $k > 3$, we show how to construct a chain of clauses in $\psi$. We introduce two sets of fresh variables $\{a_1, a_2, a_3, \ldots, a_{k-1}\}$ for $T$ and $\{b_1, b_2, b_3, \ldots, b_{k-1}\}$ for $F$ as follows:

Each clause of $\varphi$ gets separate sets of fresh variables for its chain, and we let $A = \{a_1, a_2, a_3, \ldots\}$ for $T$ and $B = \{b_1, b_2, b_3, \ldots\}$ for $F$ be the sets of all fresh variables for all chains. Finally we get a 5-CNF $\psi$ over sets of variables $Y_T = X_T \cup A$ and $Y_F = X_F \cup B$. We claim that $T$ has a winning strategy in $(\varphi, X_T, X_F)$ if $T$ has a winning strategy in $(\psi, Y_T, Y_F)$. 

### Algorithm 1: $T$'s winning strategy in $(\psi, Y)$ when $T$ has a winning strategy in $(\varphi, X)$.

1. while there is a remaining $X$-variable do
2. if (first move) or (F played an $X$-variable in the previous move) then
3. play according to the same winning strategy as in $(\varphi, X)$
4. else if $F$ played $a_i$ or $b_i$ in the previous move then play the other one to make $a_i \neq b_i$
5. while there is a remaining $A$-variable or $B$-variable do
6. if (F played $a_i$ or $b_i$ in the previous move) and (one of $a_i$ or $b_i$ remains unplayed) then
7. play the other one to make $a_i \neq b_i$
8. else pick a remaining $a_i$ or $b_i$ and assign it $z_i$'s value from Claim 1

### Algorithm 2: $F$'s winning strategy in $(\psi, Y)$ when $F$ has a winning strategy in $(\varphi, X)$.

1. while there is a remaining variable do
2. if $T$ played an $X$-variable in the previous move then
3. play according to the same winning strategy as in $(\varphi, X)$
4. else if $T$ played $a_i$ or $b_i$ in the previous move then play the other one to make $a_i = b_i$
In Section 2, we analyze the complexity of the games $G$ unsatisfied and to Claim 1, for all assignments $p$.

If $(\varphi, X_T, X_F)$ as Algorithm 3. To see that the strategy works, note that the winning strategy in $(\varphi, X_T, X_F)$ ensures that $\varphi(x)$ is satisfied by the assignment $x$ to $X_T \cup X_F$ in the first phase, so according to Claim 1, there is an assignment $z$ to $Z$ such that $\varphi'(x, z)$ is satisfied. $T$ can ensure that for each $i$, either $a_i = z_i$ or $b_i = z_i$ (since $a_i = z_i$ due to line 8, or $a_i \neq b_i$ due to line 4 or line 7) and thus $\psi(x, a, b)$ gets satisfied, since $\varphi'(x, z)$ is satisfied and each clause of $\psi$ is identical to a clause from $\varphi'$ but with each $z_i$ replaced with $a_i \lor b_i$ and $\overline{z}_i$ replaced with $\overline{a}_i \lor \overline{b}_i$.

Suppose $F$ has a winning strategy in $(\varphi, X_T, X_F)$. We describe $F$’s winning strategy in $(\varphi, Y_T, Y_F)$ as Algorithm 4. To see that the strategy works, note that the winning strategy in $(\varphi, X_T, X_F)$ ensures that $\varphi(x)$ is unsatisfied by the assignment $x$ to $X_T \cup X_F$, so according to Claim 1, for all assignments $z$ to $Z$, $\varphi'(x, z)$ is unsatisfied. $F$ can ensure that for each $i$, $a_i = b_i$; let us call this common value $z_i$. Thus $\psi(x, a, b)$ is unsatisfied, since $\varphi'(x, z)$ is unsatisfied and $\psi(x, a, b) = \varphi'(x, z)$.

### 3 2-CNf

In order to analyze the complexity of the games $G_2$ and $G_2^\%$, we construct a directed graph $g(\varphi, X)$ by the classical technique for 2-SAT:

- For each variable $x_i \in X$, form two nodes $x_i$ and $\overline{x}_i$. Let $\ell_i$ refer to either $x_i$ or $\overline{x}_i$.\(^2\)
- For each clause $(\ell_i \lor \ell_j)$, add two directed edges $\overline{\ell}_i \rightarrow \ell_j$ and $\overline{\ell}_j \leftarrow \ell_i$. In case of a single variable clause $(\ell_i)$, consider the clause as $(\ell_i \lor \ell_i)$ and add one directed edge $\overline{\ell}_i \rightarrow \ell_i$.\(^2\)

\(^2\) In Section 2, $\ell_i$ represented an arbitrary literal; in Section 3, $\ell_i$ always represents either $x_i$ or $\overline{x}_i$. 

---

**Algorithm 3:** $T$’s winning strategy in $(\psi, Y_T, Y_F)$ when $T$ has a winning strategy in $(\varphi, X_T, X_F)$.

```
1 while there is a remaining $X_T$-variable do
2   if (first move) or ($F$ played an $X_F$-variable in the previous move) then
3     play according to the same winning strategy as in $(\varphi, X_T, X_F)$
4   else if $F$ played $b_i$ in the previous move then play $a_i$ to make $a_i \neq b_i$
5 while there is a remaining $A$-variable do
6   if ($F$ played $b_i$ in the previous move) and ($a_i$ remains unplayed) then
7     play $a_i$ to make $a_i \neq b_i$
8   else pick a remaining $a_i$ and assign it $z_i$’s value from Claim 1
```

**Algorithm 4:** $F$’s winning strategy in $(\psi, Y_T, Y_F)$ when $F$ has a winning strategy in $(\varphi, X_T, X_F)$.

```
1 while there is a remaining variable do
2   if $T$ played an $X_T$-variable in the previous move then
3     play according to the same winning strategy as in $(\varphi, X_T, X_F)$
4   else if $T$ played $a_i$ in the previous move then play $b_i$ to make $a_i = b_i$
```

In the graph, every path $\ell_i \rightsquigarrow \ell_j$ has a mirror path $\overline{\ell}_i \leftarrow \overline{\ell}_j$. If there exist two paths $\ell_i \rightsquigarrow \ell_j$ and $\ell_i \rightsquigarrow \ell_k$, we express this as $\ell_i \rightsquigarrow \ell_j \rightsquigarrow \ell_k$. We are interested in strongly connected components, which we call strong components for short.

The 2-CNF game analogy on this graph is, if any variable $x_i$ is assigned a bit value in $\varphi$, then in the graph both nodes $x_i$ and $\overline{x}_i$ are assigned. Conversely, if say a player assigns a bit value to a node $\ell_i$, then the complement node $\overline{\ell}_i$ simultaneously gets assigned the opposite value. If $\ell_i$ refers to $x_i$, then $x_i$ gets assigned the same value as $\ell_i$ and similarly for $\overline{\ell}_i$ referring to $\overline{x}_i$. Thus we can describe strategies as assigning bit values to nodes in the graph.

In a satisfying assignment for $\varphi$, there must not exist any false implication edge $(1 \rightarrow 0)$ in the graph. In fact, the graph must not have any path $(1 \rightarrow 0)$ since the path will contain at least one $(1 \rightarrow 0)$ edge. Player F’s goal is to create a false implication and player T will try to make all implications true.

We prove Theorem 3 in Section 3.1 and Theorem 4 in the full version.

### 3.1 $G_2$

$G_2$ is the unordered analogue of the 2-TQBF game. We prove Theorem 3 by separately considering the cases $G_{2,F\ldots F}$ in Section 3.1.1, $G_{2,F\ldots T}$ in Section 3.1.2, and $G_{2,T\ldots}$ in Section 3.1.3.

#### 3.1.1 $G_{2,F\ldots F} \in \text{Linear Time}$

**Lemma 5.** F has a winning strategy in $G_{2,F\ldots F}$ iff at least one of the following statements holds in the graph $g(\varphi, X)$:

1. There exists a node $\ell_i$ such that $\overline{\ell}_i \rightsquigarrow \ell_i$.
2. There exist three nodes $\ell_i, \ell_j, \ell_k$ such that $\ell_j \rightsquigarrow \ell_i \rightsquigarrow \ell_k$.
3. There exist two nodes $\ell_i, \ell_j$ such that $\ell_i \rightsquigarrow \ell_j$.

**Proof.** Suppose at least one of the statements holds.

If statement (1) holds, F can win by $\ell_i = 0$ as the very first move.

If statement (2) holds but statement (1) does not, there can be two cases:

- In the first case, $\ell_i, \ell_j, \ell_k$ represent three distinct variables. At the beginning, F can play $\ell_i = 0$, then whatever T plays, F still has at least one of $\ell_j$ or $\ell_k$ to play. F can assign $\ell_j$ or $\ell_k$ as 1 and wins.

- In the second case, $\ell_i, \ell_j, \ell_k$ do not represent three distinct variables. The only possibility is that $\ell_k$ is $\overline{\ell}_j$, i.e., $\ell_j \rightsquigarrow \ell_i \rightsquigarrow \overline{\ell}_j$. F can play $\ell_i = 0$, then whatever the value of $\ell_j$, F wins.

If statement (3) holds but statement (1) does not, F can wait by playing variables other than $x_i, x_j$ with arbitrary values until T plays $x_i$ or $x_j$. Then F can immediately respond by making $\ell_i \neq \ell_j$ and win. As F moves last, he/she can always wait for that opportunity.

Conversely, suppose none of the statements hold. Then we claim the graph has no two edges that share an endpoint. Otherwise, two edges that share an endpoint would cause statement (2) or statement (3) to be satisfied. We show this by considering all possible ways of two edges sharing an endpoint:

- $\ell_i \rightsquigarrow \ell_j$: Satisfies statement (3).
- $\ell_j \rightarrow \ell_i \rightarrow \ell_k$ or its mirror $\overline{\ell}_j \leftarrow \overline{\ell}_i \rightarrow \overline{\ell}_k$: Satisfies statement (2).
- $\ell_k \rightarrow \ell_j \rightarrow \ell_i$: Satisfies statement (2).
Figure 1 T has a winning strategy in $G_{2,F\ldots F}$ for $(\overline{\varphi} \lor x_3) \land (x_4 \lor x_5)$.

Algorithm 5: Linear Time Algorithm for $G_{2,F\ldots F}$.

1. construct $g(\varphi, X)$
2. foreach $x_i \in X$ do
3. if $(x_i \rightarrow \overline{x}_i)$ or $(x_i \leftarrow \overline{x}_i)$ or ($x_i$ has at least two incident edges) then output F
4. output T

So, the graph can only have some isolated nodes and isolated edges. Since statement (1) does not hold, there are no edges between complementary nodes. An example of such a graph looks like Figure 1. Conversely, in any such graph (like Figure 1) none of statements (1), (2), (3) holds.

Now, we describe a winning strategy for T on such a graph. If F plays $\ell_i$ or $\ell_j$ of any fresh (both endpoints unassigned) edge $\ell_i \rightarrow \ell_j$, T plays in the same edge by the same bit value for the other node, i.e., making $\ell_i = \ell_j$. Otherwise, T picks any remaining node $\ell_i$. If $\ell_i$ is isolated, T assigns any arbitrary bit value. If $\ell_i$ has an incoming edge, T plays $\ell_i \rightarrow 1$. If $\ell_i$ has an outgoing edge, T plays $\ell_i \leftarrow 0$.

The strategy works, since all the edges $\ell_i \rightarrow \ell_j$ will be satisfied, by either $\ell_i = \ell_j$ or $\ell_i = 0$ or $\ell_j = 1$.

The characterization of such a graph in the proof of Lemma 5 can be verified in linear time, and that yields a Linear Time algorithm for $G_{2,F\ldots F}$. Details of the idea have been described as Algorithm 5.

3.1.2 $G_{2,F\ldots T} \in$ Linear Time

The characterization is the same as for $G_{2,F\ldots F}$ but without statement (3).

Lemma 6. F has a winning strategy in $G_{2,F\ldots T}$ iff at least one of the following statements holds in the graph $g(\varphi, X)$:

1. There exists a node $\ell_i$ such that $\overline{\ell}_i \leftarrow \overline{\ell}_i$.
2. There exist three nodes $\ell_i$, $\ell_j$, $\ell_k$ such that $\overline{\ell}_j \leftarrow \overline{\ell}_i \leftarrow \overline{\ell}_k$.

Proof. Suppose one of the statements holds. In Lemma 5, we have already seen that statement (1) and statement (2) allow player F to win at the beginning.

Conversely, suppose none of the statements hold. The graph can have strong components of size 2. Other than that, there are no two edges sharing an endpoint because statement (2) does not hold. So, the graph can only have some isolated nodes, isolated edges, and isolated strong components of size 2. Since statement (1) does not hold, there are no edges between complementary nodes. An example of such a graph looks like Figure 2. Conversely, in any such graph (like Figure 2) none of statements (1), (2) holds.
Algorithm 6: Linear Time Algorithm for $G_{2,F\ldots T}$.

1. construct $g(\varphi, X)$
2. foreach $x_i \in X$ do
3.   if $(x_i \rightarrow \varpi_i)$ or $(x_i \leftarrow \varpi_i)$ or $(x_i$ has at least two neighbors) then output $F$
4.   output $T$

Now, we describe a winning strategy for $T$ on such a graph. If $F$ plays $\ell_i$ or $\ell_j$ of any fresh (both endpoints unassigned) edge $\ell_i \rightarrow \ell_j$ or strong component $\ell_i \leftrightarrow \ell_j$, $T$ plays in the same edge or strong component by the same bit value for the other node, i.e., making $\ell_i = \ell_j$. Otherwise, $T$ picks any remaining isolated node and gives it any arbitrary bit value. Since $|X|$ is even, $T$ can always play such a node.

The strategy works, since all the edges $\ell_i \rightarrow \ell_j$ will be satisfied by $\ell_i = \ell_j$.

The characterization of such a graph in the proof of Lemma 6 can be verified in linear time, and that yields a Linear Time algorithm for $G_{2,F\ldots T}$. Details of the idea have been described as Algorithm 6.

3.1.3 $G_{2,T\ldots}$ ∈ Linear Time

In order to win $G_{2,T\ldots}$, at the beginning $T$ must locate a node $\ell_i$ such that after playing it, the game is reduced to a $G_{2,F\ldots}$ game such that $T$ still has a winning strategy in it. So, $T$’s success depends on finding such a node $\ell_i$. On the other hand, $F$’s success depends on there not existing such a node $\ell_i$.

Lemma 7. $T$ has a winning strategy in $G_{2,T\ldots}$ iff there exists an $\ell_i$ with no outgoing edges such that after deleting $\ell_i$, $\varpi_i$ and their incident edges, in the rest of the graph $T$ has a winning strategy in $G_{2,F\ldots}$.

Proof. Suppose $T$ has a winning strategy in $G_{2,T\ldots}$. Let $T$’s first move in the winning strategy be $\ell_i = 1$ (or $\ell_i = 0$). Then $\ell_i$ must not have any outgoing edge, otherwise either that edge goes to $\bar{\ell_i}$ or $F$ could play the other endpoint node of that edge as 0 and win.

Conversely, suppose there exists such an $\ell_i$. At the beginning, $T$ can play $\ell_i = 1$, and all the incoming edges to $\ell_i$ and outgoing edges from $\bar{\ell_i}$ get satisfied. Then $T$ can continue the game according to the winning strategy in $G_{2,F\ldots}$ for the rest of the graph and win. For example, in Figure 3, $T$’s winning strategy is to play $\ell_i = 1$ at the beginning then continue the winning strategy for $G_{2,F\ldots}$.

We define $L$ as the set of all nodes that have no outgoing edges. If $|L| = 0$, then according to Lemma 7, $T$ has no winning strategy in $G_{2,T\ldots}$. If $|L| > 0$, then the trivial algorithm for $G_{2,T\ldots}$ is, checking for each node $\ell_i \in L$, whether or not after playing $\ell_i = 1$ the rest of the graph becomes a winning graph for $T$ in $G_{2,F\ldots}$, i.e., running Algorithm 5 or Algorithm 6 for $O(|L|)$ times, which is a quadratic time algorithm. We argue that we can do better than that.
We filter the possibilities in \( L \) and show that there are only three cases to consider:

- There exists a node \( \ell_i \in L \) such that statement (1) from Lemma 5 and Lemma 6 holds. We consider this case in Claim 2.
- There exists a node \( \ell_i \in L \) such that statement (2) from Lemma 5 and Lemma 6 holds. We consider this case in Claim 3.
- There exists no node \( \ell_i \in L \) such that statement (1) or statement (2) from Lemma 5 and Lemma 6 holds. We consider this case in Claim 4.

Then in Claim 5 and Claim 6 we analyze the efficiency of this approach. We will provide proofs of Claim 2 to Claim 6 in the full version.

\begin{figure}[h]
\centering
\includegraphics[width=0.4\textwidth]{twin}\n\caption{T’s winning graph in \( G_{2,T} \) (all edges incident to \( \ell_i \) or \( \tilde{\ell}_i \) are optional).}
\end{figure}

\begin{itemize}
\item **Claim 2.** If there exists \( \ell_i \in L \) such that \( \tilde{\ell}_i \rightarrow \rightarrow \ell_i \) and T has a winning strategy in \( G_{2,T} \), then T’s first move must be \( \ell_i = 1 \).
\item **Claim 3.** If there exists \( \ell_i \in L \) such that \( \ell_j \rightarrow \rightarrow \ell_i \leftrightarrow \ell_k \) for two other nodes \( \ell_j, \ell_k \) and T has a winning strategy in \( G_{2,T} \), then T’s first move must be \( \ell_i = 1 \) or \( \tilde{\ell}_j = 1 \) or \( \tilde{\ell}_k = 1 \).
\item **Claim 4.** If there exists no \( \ell_i \in L \) such that \( \tilde{\ell}_i \rightarrow \rightarrow \ell_i \leftrightarrow \ell_j \leftrightarrow \ell_k \) for two other nodes \( \ell_j, \ell_k \) and T has a winning strategy in \( G_{2,T} \), then for all \( \ell_i \in L \), T has a winning strategy in \( G_{2,T} \) beginning with \( \ell_i = 1 \).
\end{itemize}

The overall idea is: If we can find an \( \ell_i \) for which statement (1) or statement (2) from Lemma 5 and Lemma 6 holds, then Claim 2 and Claim 3 allow us to narrow down T’s first move to \( O(1) \) possibilities. If we cannot find such an \( \ell_i \), then Claim 4 allows T to play any arbitrary \( \ell_i \in L \) as the first move because all of them are equivalent as the first move. We define \( L^* \) as the \( O(1) \) possibilities in \( L \). Then we can run Algorithm 5 or Algorithm 6 for \( |L^*| = O(1) \) times.

In the following two claims, we show how we can efficiently verify whether or not there exists such an \( \ell_i \) for which statement (1) or statement (2) from Lemma 5 and Lemma 6 holds.

\begin{itemize}
\item **Claim 5.** There exists a constant-time algorithm for: given \( \ell_i \), find two other nodes \( \ell_j, \ell_k \) such that \( \ell_j \rightarrow \rightarrow \ell_i \leftrightarrow \ell_k \) or determine they do not exist.
\item **Claim 6.** There exists a constant-time algorithm for: given \( \ell_i \) for which there are no \( \ell_j \), \( \ell_k \) as in Claim 5, decide whether there exists a path \( \tilde{\ell}_i \rightarrow \rightarrow \ell_i \).
\end{itemize}

Now combining the whole idea from Claim 2 to Claim 6 we can develop an algorithm for \( G_{2,T} \). Details of the idea have been described as Algorithm 7.
Algorithm 7: Linear Time Algorithm for $G_{2,T...}$

1. construct $g(\varphi, X)$
2. let $L = \{\}$, $L^* = \{\}$
3. foreach node $\ell_i$ do
4.   if $\ell_i$ has no outgoing edges then $L = L \cup \{\ell_i\}$
5.   if $|L| = 0$ then output $F$
6.   foreach $\ell_i \in L$ do
7.     if $\ell_j \leadsto \ell_i \leadsto \ell_k$ for two other nodes $\ell_j, \ell_k$ (using Claim 5) then
8.       $L^* = L \cap \{\ell_i, \ell_j, \ell_k\}$ (Claim 3), break loop
9.     else if $\overline{\ell_i} \leadsto \ell_i$ (using Claim 6) then $L^* = \{\ell_i\}$ (Claim 2), break loop
10.    if $|L^*| = 0$ then $L^* = \{\ell_i\}$ for an arbitrary $\ell_i \in L$ (Claim 4)
11.   foreach $\ell_i \in L^*$ do
12.     form graph $g'$ from $g(\varphi, X)$ by deleting nodes $\ell_i, \overline{\ell_i}$ and their incident edges
13.     run Algorithm 5 or Algorithm 6 on $g'$ as the $G_{2,F...}$ game
14.   if $T$ has a winning strategy in $G_{2,F...}$ then output $T$
15. output $F$

References

Half-Duplex Communication Complexity

Kenneth Hoover
University of California San Diego, USA
khooveri@eng.ucsd.edu

Russell Impagliazzo
University of California San Diego, USA
russell@cs.ucsd.edu

Ivan Mihajlin
University of California San Diego, USA
ivmihajlin@gmail.com

Alexander V. Smal
St. Petersburg Department of Steklov Mathematical Institute of Russian Academy of Sciences, Russia
smal@pdmi.ras.ru

Abstract

Suppose Alice and Bob are communicating in order to compute some function \( f \), but instead of a classical communication channel they have a pair of walkie-talkie devices. They can use some classical communication protocol for \( f \) where in each round one player sends a bit and the other one receives it. The question is whether talking via walkie-talkie gives them more power? Using walkie-talkies instead of a classical communication channel allows players two extra possibilities: to speak simultaneously (but in this case they do not hear each other) and to listen at the same time (but in this case they do not transfer any bits). The motivation for this kind of a communication model comes from the study of the KRW conjecture. We show that for some definitions this non-classical communication model is, in fact, more powerful than the classical one as it allows to compute some functions in a smaller number of rounds. We also prove lower bounds for these models using both combinatorial and information theoretic methods.

2012 ACM Subject Classification  Theory of computation → Communication complexity

Keywords and phrases  communication complexity, half-duplex channel, information theory

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.10

Related Version A full version of the paper is available at [6], https://eccc.weizmann.ac.il/report/2018/089.

Acknowledgements We want to thank the anonymous reviewers whose careful reading and comments helped us to improve the paper.

1 Introduction

In the classical communication complexity model introduced by Yao [11] two players, Alice and Bob, are trying to compute \( f(x, y) \), for some function \( f \), where Alice knows only \( x \) and Bob knows only \( y \). Alice and Bob can communicate by sending bits to each other, one bit per round. The essential property of this classical model is that in every round of communication one player sends some bit and the other one receives it.
We define three new communication models that generalize the classical one and resemble communication over so-called half-duplex channels. A well-known example of half-duplex communication is talking via walkie-talkie: one has to hold a “push-to-talk” button to speak to another person, and one has to release it to listen. If two persons try to speak simultaneously then they do not hear each other. We consider communication models where players are allowed to speak simultaneously. Every round each player chooses one of three actions: send 0, send 1, or receive. There are three different types of rounds. If one player sends some bit and the other one receives then communication works like in the classical case, we call such rounds normal. If both players send bits during the round then these bits get lost (the same happens if two persons try to speak via walkie-talkie simultaneously), we call these rounds spent. If both players receive, we call these rounds silent. We distinguish three possible models, based on what happens in silent rounds. If in silent rounds both players receive 0, i.e., players cannot distinguish a silent round from a normal round where the other player sends 0, we call this model half-duplex communication with zero. A somewhat similar model was studied in [3] for multi-party communication with the noisy broadcast channel. Two other models, we will define later.

In this paper, we study the communication complexity of Boolean functions that are hard in the classical case. It is important to note that we care about multiplicative constants. Every classical communication can be viewed as half-duplex communication with zero and every half-duplex communication with zero can be simulated with classical communication doubling the number of rounds (see Theorem 6 and 7). So the complexity of half-duplex communication is sandwiched between the complexity of the classical case and a half of it. The task of this study is to improve these bounds.

### 1.1 Motivation

The original motivation to study these kinds of communication models arose from the question of the complexity of Karchmer-Wigderson games [8] for multiplexers. The Karchmer-Wigderson game for a function \( f : \{0,1\}^n \rightarrow \{0,1\} \) (KW game) is a (classical) communication problem where Alice is given \( x \in f^{-1}(0) \), Bob is given \( y \in f^{-1}(1) \), and they want to find an \( i \in [n] \) such that \( x_i \neq y_i \). Let \( D(KW(f)) \) be a minimal number of rounds that is enough to the KW game for \( f \) on any pair of possible inputs.

> **Conjecture 1 (KRW conjecture [7]).** Let \( f : \{0,1\}^n \rightarrow \{0,1\} \) and \( g : \{0,1\}^m \rightarrow \{0,1\} \) be Boolean non-constant functions. Then \( D(KW(g \circ f)) \approx D(KW(g)) + D(KW(f)) \), where \( g \circ f \) denotes a composition \( g \circ f : ((\{0,1\}^n)^m \rightarrow \{0,1\}) \) is defined by \( (g \circ f)(x_1, \ldots, x_m) = g(f(x_1), \ldots, f(x_m)) \) where \( x_1, \ldots, x_n \in \{0,1\} \).

This conjecture implies a super-logarithmic formula depth lower bound (and hence a super-polynomial size lower bound): we can start with a maximally hard function on \( n \) variables that requires \( \log n \) depth and construct a formula on \( n \) variables that requires super-logarithmic depth. In attempt to prove it a lot of work has been done studying KW games where one or both functions are replaced with universal relations [5, 2, 4]. Another approach to resolving the conjecture lies in examining multiplexer functions. A multiplexer (or indexing function) is a function \( M_n : \{0,1\}^2 \times \{0,1\}^n \rightarrow \{0,1\} \), such that \( M_n(t,i) = t[i] \), i.e., \( M_n \) interprets the first part of its input as the truth table of some function \( f : \{0,1\}^n \rightarrow \{0,1\} \) and the second part as an input \( x \) to the function, and outputs \( f(x) \). Multiplexers are similar to universal relations in the sense that there is a natural reduction from a KW game for some function \( f : \{0,1\}^n \rightarrow \{0,1\} \) to a KW game for multiplexer \( M_n \); if Alice and Bob are given \( x \) and \( y \) in the game for \( f \) we give them \((tt(f), x)\) and \((tt(f), y)\), respectively, in the
game for $M_n$, where $tt(f)$ is a truth table of function $f$. On the other hand, multiplexers are functions, not relations, so proving analogous results for multiplexers would be one step toward proving the KRW conjecture. Unfortunately, all the techniques that were used for universal relations cannot be applied directly to multiplexers because it is impossible to give Alice and Bob the same input string; all these techniques exploited the symmetry of universal relations that allows giving players the same input string, but this is impossible for functions because inputs of Alice and Bob come from disjoint sets.

Suppose now that Alice and Bob are solving the KW game for multiplexer $M_n$: Alice is given $(tt(f), x), x \in f^{-1}(0)$, and Bob is given $(tt(g), y), y \in g^{-1}(1)$. If the players are also given a promise that $f = g$ (note that $f$ and $g$ are parts players inputs, so Alice and Bob plays KW game for $M_n$ on a subset of inputs) then they can use a protocol for KW game for $f$. However, what if they do not have such a promise (i.e., all inputs are possible, in particular, such that $f \neq g$)? Alice can still try to act as if she plays KW game for $f$, Bob at the same time can try to act as if he plays KW game for $g$, but if in fact $f \neq g$ then in some round of this “mixed” protocol they might both want to send or both want to receive at the same time. Such protocol “mixing” is impossible in the classical model. To make it possible we extend the communication model by allowing players to speak or listen simultaneously. How does it affect the communication complexity? When answering this question we care about multiplicative constants – if in this model all (hard) functions become two times easier than in the classical case then this model is useless for proving the KRW conjecture. As a first step toward answering this question, we study the half-duplex communication complexity of Boolean functions $\{0, 1\}^n \times \{0, 1\}^n \rightarrow \{0, 1\}$.

1.2 Organization of this paper

In Section 2, we give definitions for the new communication models. Then, in Section 3, we prove trivial upper and lower bounds that follow immediately from the definitions. Next, in Section 4, we discuss methods for proving communication complexity lower bounds. In Sections 5, 6 and 7, we present our main results, upper and lower bounds for proposed communication models. Finally, in Section 8, we state several open questions.

2 Definitions

Definition 1. Let $X$, $Y$, and $Z$ be some finite sets. We say that two players, Alice and Bob, are solving the half-duplex communication problem for a relation $R \subseteq X \times Y \times Z$ if sets $X$, $Y$, $Z$, and the relation $R$ are known by both players, Alice is given some $x \in X$, Bob is given some $y \in Y$, and players want to find some $z \in Z$ such that $(x, y, z) \in R$, by communicating to each other via a half-duplex channel. The communication is organized into rounds. At each round, both players decide (depending only on their inputs and previous communication) to do one of three available actions: send 0, send 1 or receive. If one player sends some bit $b \in \{0, 1\}$ and the other one receives then the latter gets bit $b$, we call such rounds normal. If both players send bits at the same time then these bits get lost, we call such rounds spent (it is crucial that the player that is sending cannot distinguish whether this round is normal or spent). If both players receive at the same time, we call such rounds silent. There are three variants of half-duplex communication problem depending on how silent rounds work.

- In a silent round both players receive a special symbol silence, so it is possible for both players to distinguish a silent round from a normal one, the corresponding problem is called half-duplex communication problem with silence.
In a silent round both players receive 0, i.e., players cannot distinguish a silent round from a normal round where the other player sends 0, the corresponding problem is called half-duplex communication problem with zero;

In a silent round each player receives some arbitrary bit, not necessarily the same as the other player; the corresponding problem is called half-duplex communication problem with adversary.

We say that half-duplex communication problem for $R$ is solved if at the end of communication both players know some $z$, such that $(x, y, z) \in R$.

Next, we define a notion of communication protocol. In the classical case, a protocol is a binary rooted tree that describes communication of players on all possible inputs: every internal node corresponds to a state of communication and defines which of players is sending this round. Unlike the classical case in half-duplex communication player does not always know what the other’s player action was – the information about it can be “lost,” i.e., in spent rounds player do not know what the other player’s action was. It means that a player might not know what node of the protocol corresponds to the current state of communication. Note also that solving half-duplex communication problem with zero there is no need to send zeros – player can receive instead and the other player will not notice the difference. Keeping all this in mind, we give the following definition of half-duplex protocol.

Definition 2. Half-duplex communication protocol with silence that solves a relation $R \subseteq X \times Y \times Z$ is a pair $(T_A, T_B)$ of rooted trees that describe how Alice and Bob communicate on all possible inputs $(x, y) \in X \times Y$. Every node of $T_A$ corresponds to a state of Alice, every node of $T_B$ to a state of Bob. Every leaf $l$ is labeled with $z_l \in Z$. Let $\mathcal{A} = \{send(0), send(1), receive\}$ be the set of possible actions, and $\mathcal{E} = \{send(0), send(1), receive(0), receive(1), silence\}$ be the set of all possible events. Every node $v$ of $T_A$ and $(v)$ of $T_B$ is labeled with two functions $g_v : X \rightarrow \mathcal{A}$ ($g_v : Y \rightarrow \mathcal{A}$) and $h_v : \mathcal{E} \rightarrow C(v)$, where $C(v)$ is a set of child nodes of $v$. Root nodes of $T_A$ and $T_B$ correspond, respectively, to the initial states of Alice and Bob. If Alice (Bob) is in a state that corresponds to node $v \in T_A$ ($v \in T_B$), then she does action $g_v(x)$ (he does action $g_v(y)$). Events of both players are defined in a natural way by their actions in this round. The next node of the protocol is defined by the function $h$. When players reach a leaf they stop (they always reach a leaf simultaneously). The protocol is correct if for every input pair $(x, y, z) \in X \times Y$ communication ends in a pair of leaves labeled with the same $z \in Z$ such that $(x, y, z) \in R$.

Half-duplex communication protocol with zero is defined in the same way with a different set of possible events $\mathcal{E} = \{send(1), receive(0), receive(1)\}$, i.e it does not include send(0).

Half-duplex communication protocol with adversary that solves a relation $R \subseteq X \times Y \times Z$ is a pair $(T_A, T_B)$ of rooted trees that describe how Alice and Bob communicate on all possible inputs $(x, y) \in X \times Y$ and for any strategy of adversary $w \in \{0, 1\}^*$. The structure of the protocol is the same as in half-duplex communication protocol with zero, but with $\mathcal{E} = \{send(0), send(1), receive(0), receive(1)\}$. If both players decide to receive in round $i$, then Alice and Bob receive bits $w_{2i-1}$ and $w_{2i}$, respectively. The protocol is correct if for every input pair $(x, y, z) \in X \times Y$ and any strategy of adversary $w \in \{0, 1\}^*$ communication ends in two leaves labeled with the same $z \in Z$ such that $(x, y, z) \in R$.

For each of these models, a partial transcript after $k$ rounds is a pair $(\pi_a, \pi_b)$ of length-$k$ sequences over $\mathcal{E}$ that lists the events observed by Alice and Bob, respectively, after running some protocol on a pair of inputs for $k$ rounds.

The cardinality of set $\mathcal{E}$ upper bounds arity of trees $T_A$ and $T_B$: arity is 5 for half-duplex communication with silence, 3 for half-duplex communication with zero, and 4 for half-duplex communication with the adversary.
Definition 3. Half-duplex communication protocol solves a communication problem for function \( f : X \times Y \rightarrow Z \) if it solves a relation \( R(f) = \{(x, y, f(x, y)) | x \in X, y \in Y \} \).

The classical communication complexity of a communication problem for function \( f \), \( D(f) \), is defined in terms of the minimal depth of a protocol solving it. Analogously, we define communication complexity for half-duplex communication problems.

Definition 4. The minimal depth of a communication protocol solving half-duplex communication problem for function \( f \) with silence, with zero, with adversary, defines half-duplex communication complexity of function \( f \) with silence, denoted \( D^{hd}(f) \), with zero, denoted \( D^{hd}_0(f) \), with adversary, denoted \( D^{hd}_a(f) \), respectively. Analogously, we define half-duplex communication complexity of relation \( R \) with silence, \( D^{hd}_s(R) \), with zero, \( D^{hd}_0(R) \), and with adversary, \( D^{hd}_a(R) \).

In this paper we study half-duplex communication complexity for a special case of Boolean functions \( \{0, 1\}^n \times \{0, 1\}^n \rightarrow \{0, 1\} \) (i.e., \( X = Y = \{0, 1\}^n, Z = \{0, 1\} \)).

Definition 5.

- Equality function \( \text{EQ}_n : \{0, 1\}^n \times \{0, 1\}^n \rightarrow \{0, 1\} \), such that \( \text{EQ}_n(x, y) = 1 \iff x = y \).
- Inner product function \( \text{IP}_n : \{0, 1\}^n \times \{0, 1\}^n \rightarrow \{0, 1\} \), such that \( \text{IP}_n(x, y) = \bigoplus_{i \in [n]} x_i y_i \).
- Disjointness function \( \text{DISJ}_n : \{0, 1\}^n \times \{0, 1\}^n \rightarrow \{0, 1\} \), such that \( \text{DISJ}_n(x, y) = 1 \iff \forall i : x_i \neq 1 \lor y_i \neq 1 \).

All these function require \( n \) bits of communication in the classical model.

3 Trivial bounds

As far as half-duplex communication generalizes classical communication the following upper bound is immediate.

Theorem 6. For every function \( f : \{0, 1\}^n \times \{0, 1\}^n \rightarrow \{0, 1\} \), \( D^{hd}_s(f) \leq D^{hd}_0(f) \leq D^{hd}_a(f) \leq D(f) \).

Proof. Every classical communication protocol can be embedded in half-duplex communication protocol that does not use spent and silent rounds.

Next theorem shows that one can always transform half-duplex protocol with zero or with the adversary into a classical communication protocol of double depth.

Theorem 7. For every function \( f : \{0, 1\}^n \times \{0, 1\}^n \rightarrow \{0, 1\} \), \( \frac{D(f)}{2} \leq D^{hd}_0(f) \leq D^{hd}(f) \).

Proof. Every t-round half-duplex communication protocol with zero or with the adversary can be transformed into 2t-round classical communication protocol. Every round of the original protocol corresponds to two consecutive rounds of the new one: on the first round Alice sends a bit she was sending in the original protocol or sends 0 if she was receiving, at second round Bob does the same thing.

As we will see later, half-duplex protocols with silence can use silent rounds as an additional third symbol and hence not every t-round half-duplex protocol with silence can be embedded in 2t classical protocol. The following theorem shows that instead, we can embed every such protocol in a classical protocol with 3t rounds.

Theorem 8. For every function \( f : \{0, 1\}^n \times \{0, 1\}^n \rightarrow \{0, 1\} \), \( D^{hd}_a(f) \geq \frac{D(f)}{3} \).
**Proof.** Every $t$-round half-duplex communication protocol with silence can be transformed into $3t$-round classical communication protocol. Every round of the original protocol corresponds to three consecutive rounds of the new one: on the first round, Alice sends 1 to indicate if she was sending a bit in the original protocol, or sends 0 otherwise, at second round Bob does the same thing symmetrically. After that, they are both aware of the intentions of each other. If they were both planning to send, they could skip the third round. If they were both planning to receive, then they can assume that they heard silence. If one player was planning to send and the other one was planning to receive they can perform such action on the third round.

**Remark.** Theorems 6, 7, and 8 holds also for $f : \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^k$.

## 4 Methods for lower bounds

### 4.1 Rectangles

Many lower bounds on classical communication complexity were proved by considering combinatorial rectangles associated with the nodes of communication protocol [10]: it is easy to see that every node $v$ of the (classical) protocol corresponds to a combinatorial rectangle $R_v = X_v \times Y_v$, where $X_v \subseteq X$, $Y_v \subseteq Y$, such that if Alice and Bob are given an input from $R_v$ then their communication will necessarily pass through node $v$. This implies that the rectangles associated with the child nodes of $v$ define a subdivision of $R_v$.

There is a general technique [10] for proving lower bounds using associated combinatorial rectangles in: if for some sub-additive measure $\mu$ defined on combinatorial rectangles we show both a lower bound on the measure of $X \times Y$, the rectangle in the root node, i.e., $\mu(X \times Y) \geq \mu_v$ for some $\mu_v > 0$, and an upper bound on the measure of rectangles in leaves, i.e., for every leaf $l$ the measure of the corresponding rectangle $R_l$ is at most $\mu_l$ for some $\mu_l > 0$, then we can claim lower bound of $\log(\mu_v/\mu_l)$ on the depth of the protocol.

One of the most studied sub-additive measure on rectangles is $\mu_M(R)$ that is equal to the minimal number of monochromatic rectangles that covers $R$. Rectangle $R$ is $z$-monochromatic respect to function $f$ for some $z \in Z$ if for all $(x,y) \in R$, $f(x,y) = z$. As far as both players have to come up with the same answer at the end of communication every rectangle in leaves is monochromatic, thus for this measure $\mu_l = 1$.

We can use almost the same technique for half-duplex protocols. There are some technical differences that we have to keep in mind. First of all, we can apply this idea to both trees $T_A$ and $T_B$. We should also note that trees $T_A$ and $T_B$ are non-binary; hence arity became the base of the logarithm. Secondly, we should be careful while defining associated combinatorial rectangles for half-duplex protocols with adversary – in case of silent rounds the next node of the protocol depends also on a strategy $w$ of adversary, so we have to formally consider $w$ as a part of input. This leads to the following lower bound for equality.

**Theorem 9.**
\[
\begin{align*}
D^h_a(EQ_a) & \geq \log_5 2^n = n/ \log 5, \\
D^{hd}_a(EQ_a) & \geq \log_3 2^n = n/ \log 3, \\
D^{hd}_a(EQ_a) & \geq \log_4 2^n = n/2.
\end{align*}
\]

**Proof.** Let $\mu = \mu_M$. All leaf rectangles are monochromatic, $\mu_l = 1$. Every 1-monochromatic rectangle is of size one: if some rectangle contains two elements, say $(x,x)$ and $(x',x')$, then it also contains $(x,x')$ and $(x',x)$, so it is not 1-monochromatic. Thus, the root rectangle has measure at least $\mu_v = 2^n + 1$ (see [10] for more information).
Surprisingly, as we will see later, first two result are sharp up to additive logarithmic term. We developed an extension of this technique that we call round elimination.

4.2 Round elimination

Let us fix a protocol for some half-duplex communication problem and consider the first round. Let $R_c = X \times Y$ be the corresponding rectangle of all possible inputs. We can subdivide $R_c$ in nine rectangles, one for each possible combination of actions.

<table>
<thead>
<tr>
<th>Alice</th>
<th>Bob</th>
<th>send(0)</th>
<th>send(1)</th>
<th>receive</th>
</tr>
</thead>
<tbody>
<tr>
<td>send(0)</td>
<td>$R_{00}$</td>
<td>$R_{01}$</td>
<td>$R_{0r}$</td>
<td></td>
</tr>
<tr>
<td>send(1)</td>
<td>$R_{10}$</td>
<td>$R_{11}$</td>
<td>$R_{1r}$</td>
<td></td>
</tr>
<tr>
<td>receive</td>
<td>$R_{r0}$</td>
<td>$R_{r1}$</td>
<td>$R_{rr}$</td>
<td></td>
</tr>
</tbody>
</table>

Consider two rectangles: $R_{good} = R_{00} \cup R_{01} \cup R_{0r}$ and $R_{bad} = R_{0r} \cup R_{1r}$. If we restrict $f$ to be a partial function defined only on $R_{good}$, i.e., players will always get some $(x, y) \in R_{good}$, then there is no need in the first round – the information the players get about the other part of the input is fixed: Alice does not get any information, Bob can receive 0 if he decides to receive. On the other hand if we restrict $f$ to $R_{bad}$ then the first round is still needed: Bob can receive both 0 and 1 and this information in necessary to proceed to the next round.

Lest call a rectangle $R$ good for (partial) function $f$ if restricting $f$ to $R$ makes the first round unnecessary (i.e., protocol without the first round is correct for all $(x, y) \in R$). The idea of this method is to consider some covering of $R_c$ with a set of good rectangles and prove that there is always a good rectangle of large enough measure. If we can show that there is always a rectangle of measure at least $\alpha \cdot \mu(R_c)$ then we can iterate this idea and claim that protocol depth is at least $\log_{\frac{\mu_r}{\mu}}$. Fix a protocol $\mathcal{P}$. If for any rectangle $R$ appearing in the protocol there is a good subrectangle for function $f \res R$ of measure at least $\alpha \cdot \mu(R)$ then the depth of the protocol is at least $\log_{\frac{\mu_r}{\mu}}$.

**Lemma 10.** Let $\mu$ be some sub-additive measure on rectangles such that $\mu(X \times Y) \geq \mu_r$ and for any leaf rectangle $R_l$, $\mu(R_l) \leq \mu_r$. Fix a protocol $\mathcal{P}$. If for any rectangle $R$ appearing in the protocol there is a good subrectangle for function $f \res R$ of measure at least $\alpha \cdot \mu(R)$ then the depth of the protocol is at least $\log_{\frac{\mu_r}{\mu}}$.

**Proof.** We start with $R = X \times Y$. Every round we show that $f \res R$ can be restricted to some good $R_{good} \subseteq R$ such that $\mu(R_{good}) \geq \alpha \cdot \mu(R)$, let $R$ to be $R_{good}$, and proceed to the next round until we reach a leaf. Thus there are at least $\log_{\frac{\mu_r}{\mu}}$ rounds.

4.3 Upper bound on internal information

Another useful tool for proving lower bounds on the communication complexity of problems in the classical model is the upper bound on the information Alice and Bob have learned about the other’s inputs, as a function of the number of rounds.

**Definition 11.** Let $f$ be a partial function and $\mathcal{P}$ a half-duplex communication protocol computing $f$, and $\mathcal{D}$ an arbitrary distribution over the domain of $f$. Let $\mathcal{X}$, and $\mathcal{Y}$ be the marginal distributions over inputs to Alice and Bob, also, let $\Pi_A$ and $\Pi_B$ be the marginal distributions over Alice and Bob’s transcripts induced by $\mathcal{D}$. An internal information cost of protocol $\mathcal{P}$ is $\text{IC}_\mathcal{D}(\mathcal{P}) = I(\mathcal{X} : \Pi_B \condition \mathcal{Y}) + I(\mathcal{Y} : \Pi_A \condition \mathcal{X})$. For any $k$ let $\Pi_A^k$ and $\Pi_B^k$ be the marginal distributions over Alice and Bob’s partial transcripts after running $\mathcal{P}$ for $k$ rounds induced by $\mathcal{D}$. An internal information cost of first $k$ rounds of $\mathcal{P}$ is $\text{IC}_\mathcal{D}^k(\mathcal{P}) = I(\mathcal{X} : \Pi_B^k \condition \mathcal{Y}) + I(\mathcal{Y} : \Pi_A^k \condition \mathcal{X})$. 

ISAAC 2018
For more information on information theory, we refer to [1, 4]. We use this approach to prove lower bounds on the inner product using the following Lemma.

**Lemma 12.** Let \( D \) be uniform distribution over all input pairs of \( \text{IP}_n \) (pairs of \( n \)-bit strings). If any half-duplex communication protocol with silence/zero/adversary \( P \) computing \( \text{IP}_n \) and for every \( k, \text{IC}^D_k(P) \leq \alpha k \), for some \( \alpha \geq 1 \), then half-duplex complexity of \( \text{IP}_n \) with silence/zero/adversary is at least \( \frac{n}{\alpha} \).

To prove this Lemma we use the following property of \( \text{IP}_n \).

**Lemma 13.** Every leaf rectangle of a protocol for \( \text{IP}_n \) has size at most \( 2^n \).

**Proof of Lemma 12.** For uniform distribution over all input pairs \( H(X | Y) + H(Y | X) = 2n \). By Lemma 13 each leaf of any correct protocol contains at most \( 2^n \) input pairs in its rectangle, thus \( H(X | Y, \Pi_B) + H(Y | X, \Pi_A) \leq n \). If \( \text{IP}_n \) has a protocol of depth \( k \) then

\[
\alpha k \geq I(X : \Pi^B_k|Y) + I(Y : \Pi^A_k|X)
= H(X | Y) - H(X | Y, \Pi^B_k) + H(Y | X) - H(Y | X, \Pi^A_k) \geq n.
\]

\[\Box\]

## 5 Half-duplex communication with silence

The main advantage of this model over the other models we consider is that whenever players have silent round, they learn about it. In some sense they have a third symbol in the alphabet – receiving player can get either 0/1 or a special symbol corresponding to “silence”.

Next theorem shows how players can take the advantage of silence to transfer data.

**Theorem 14.** For every \( f : \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\} \), \( D^{hd}_{s}(f) \leq \lceil \frac{n}{\log 3} \rceil + 1 \).

**Proof.** Alice encodes \( x \) in ternary alphabet \( \{0,1,2\} \) and sends it to Bob: in order to send 0 or 1 Alice sends the corresponding bit, sending 2 is emulated by receiving (keeping silence). This requires \( \lceil \log_3 2^n \rceil = \lceil n/\log 3 \rceil \) bits. At the last round Bob computes \( f(x, y) \) and sends the result back to Alice.

Using the idea of non-binary encoding, we prove a better upper bound for equality.

**Theorem 15.** \( D^{hd}_{s}(\text{EQ}_n) \leq \lceil \frac{n}{\log 5} \rceil + \lceil \log n/\log 3 \rceil + 2 \).

**Proof.** Alice and Bob encode their inputs in alphabet of size five \( \{0,1,2,3,4\} \). Then they process their inputs symbol by symbol sequentially in \( \lceil n/\log 5 \rceil \) rounds. At round \( i \) they process \( i \)th symbol in the following manner.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>send(0)</td>
<td>receive</td>
</tr>
<tr>
<td>1</td>
<td>send(1)</td>
<td>receive</td>
</tr>
<tr>
<td>2</td>
<td>receive</td>
<td>send(0)</td>
</tr>
<tr>
<td>3</td>
<td>receive</td>
<td>send(1)</td>
</tr>
<tr>
<td>4</td>
<td>receive</td>
<td>receive</td>
</tr>
</tbody>
</table>

If \( i \)th round is normal then one player can check whether \( i \)th symbols are different. If \( i \)th round is silent then again one player knows if \( i \)th symbols are different. If after \( \lceil n/\log 5 \rceil \) rounds one of the players has already learned that the answer is 0, then he or she sends 0. If this round is not silent, then both players know that the answer is 0. Otherwise, Alice and Bob have to make sure that there were no spent rounds. To check it, Alice sends the number
of normal rounds she was receiving encoded in ternary, that requires \(\lceil \log n / \log 3 \rceil\) rounds. Bob checks whether this number is equal to the number of rounds he was sending in. If so, inputs are equal. In the last round, Bob sends the answer back to Alice.

Using almost the same ideas we can show an upper bound for disjointness.

\[ \text{Theorem 16.} \quad D^{hd}(\text{DISJ}_n) \leq \lceil n/2 \rceil + 2. \]

\[ \text{Proof.} \quad \text{Alice and Bob process their inputs two bits per round, } \lceil n/\log 2 \rceil \text{ rounds. At round } i \text{ they process symbols } 2i - 1 \text{ and } 2i \text{ in the following manner.} \]

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>send(0)</td>
<td>receive</td>
</tr>
<tr>
<td>01</td>
<td>receive</td>
<td>send(0)</td>
</tr>
<tr>
<td>10</td>
<td>receive</td>
<td>send(1)</td>
</tr>
<tr>
<td>11</td>
<td>receive</td>
<td>receive</td>
</tr>
</tbody>
</table>

At the end of communication Bob tells Alice whether there was a silent round in which Bob’s input was 11 (i.e., inputs are not disjoint). Alice tells Bob whether she ever received 01 or 11, or received 1 having 10 or 11 (again, inputs are not disjoint).

To prove lower bounds one can use round elimination and get the following lower bound for the inner product (see full version [6] for the proof).

\[ \text{Theorem 17.} \quad D^{hd}(\text{IP}_n) \geq n/2. \]

This lower bound can be improved using upper bound on internal information.

\[ \text{Theorem 18.} \quad D^{hd}(\text{IP}_n) \geq n/1.67. \]

\[ \text{Proof.} \quad \text{To apply Lemma 12 it is enough to show that } I(\mathcal{X}: \Pi^k_B \mid \mathcal{Y}) + I(\mathcal{Y}: \Pi^k_A \mid \mathcal{X}) \leq \alpha k, \text{ where } \alpha \leq 1.67. \text{ We will induct on } k: \text{ the number of rounds. For } k = 0, \text{ there is only one possible partial transcript for either player, the empty transcript, and thus the result is immediate. Now suppose that this is true in round } k. \text{ Let } \mathcal{E}^{k+1}_A \text{ and } \mathcal{E}^{k+1}_B \text{ be the marginal distributions over which event each player will observe. Note that} \]

\[ I(\mathcal{X}: \Pi^k_B \mid \mathcal{Y}) = I(\mathcal{X}: \Pi^k_B \mid \mathcal{E}^{k+1}_B) + I(\mathcal{X}: \Pi^k_B \mid \mathcal{Y}) + I(\mathcal{X}: \mathcal{E}^{k+1}_B \mid \mathcal{Y}, \Pi^k_B). \]

Thus, it suffices to show that

\[ I(\mathcal{X}: \mathcal{E}^{k+1}_B \mid \mathcal{Y}, \Pi^k_B) + I(\mathcal{Y}: \mathcal{E}^{k+1}_B \mid \mathcal{X}, \Pi^k_B) \leq \alpha. \]

\[ I(\mathcal{X}: \mathcal{E}^{k+1}_B \mid \mathcal{Y}, \Pi^k_B) = H(\mathcal{E}^{k+1}_B \mid \mathcal{Y}, \Pi^k_B) - H(\mathcal{E}^{k+1}_B \mid \mathcal{Y}, \Pi^k_B, \mathcal{X}) = H(\mathcal{E}^{k+1}_B \mid \mathcal{Y}, \Pi^k_B). \]

The second term here is zero because values of \( \mathcal{X} \) and \( \mathcal{Y} \) unambiguously determine the entire protocol. So it is enough to bound

\[ H(\mathcal{E}^{k+1}_B \mid \mathcal{Y}, \Pi^k_B) = E_{y,\pi}[H(\mathcal{E}^{k+1}_B \mid \mathcal{Y} = y, \Pi^k_B = \pi)]. \]

Let \( A^{k+1}_A \) and \( A^{k+1}_B \) be the marginal distributions over players actions in round \( k+1 \). Note that 

\[ A^{k+1}_B \text{ is a function of } y \text{ and } \pi. \]

If for some pair \((y, \pi)\) Bob sends, i.e., \( A^{k+1}_B = \text{send}(0) \) or \( A^{k+1}_B = \text{send}(1) \), then 

\[ H(\mathcal{E}^{k+1}_B \mid \mathcal{Y} = y, \Pi^k_B = \pi) = 0. \]

For the sake of brevity we denote \( E_{y,\pi} \) an event “\( \mathcal{Y} = y, \Pi^k_B = \pi \)” and \( r \) an action \( \text{receive} \in A \).

\[ H(\mathcal{E}^{k+1}_B \mid \mathcal{Y}, \Pi^k_B) = \Pr[A^{k+1}_B = r] \cdot H(\mathcal{E}^{k+1}_B \mid \mathcal{Y}, \Pi^k_B, A^{k+1}_B = r). \]

Notice that player’s action choices are independent, hence

\[ H(\mathcal{E}^{k+1}_B \mid \mathcal{Y}, \Pi^k_B, A^{k+1}_B = r) = H(\mathcal{E}^{k+1}_A \mid \mathcal{Y}, \Pi^k_B) \leq H(\mathcal{E}^{k+1}_A). \]
10:10  Half-Duplex Communication Complexity

This gives us the following bound.

\[ H(\mathcal{X}^{k+1} \mid \mathcal{Y}, \Pi_B^k) \leq \Pr[A_B^{k+1} = r] \cdot H(A_A^{k+1}). \]

The same argument works for \( I(\mathcal{Y} : \Pi_A^k \mid \mathcal{X}) \) and hence we get,

\[ I(\mathcal{X} : \Pi_B^k \mid \mathcal{Y}) + I(\mathcal{Y} : \Pi_A^k \mid \mathcal{X}) \leq \Pr[A_B^{k+1} = r] \cdot H(A_A^{k+1} = a) \]

\[ + \Pr[A_A^{k+1} = r] \cdot H(A_B^{k+1} = a). \]

Now let’s denote \( a_0 \) and \( a_1 \) to be the fractions of inputs for which Alice sends 0 or 1, respectively, and symmetrically \( b_0 \) and \( b_1 \) to be the fractions of inputs for which Bob sends 0 or 1, respectively. The right hand side of the above inequality can be rewritten as follows.

\[ (1 - b_0 - b_1) \cdot \left( a_0 \cdot \log_3 \frac{1}{a_0} + a_1 \cdot \log_3 \frac{1}{a_1} + (1 - a_0 - a_1) \cdot \log_3 \frac{1}{1 - a_0 - a_1} \right) \]

\[ + (1 - a_0 - a_1) \cdot \left( b_0 \cdot \log_3 \frac{1}{b_0} + b_1 \cdot \log_3 \frac{1}{b_1} + (1 - b_0 - b_1) \cdot \log_3 \frac{1}{1 - b_0 - b_1} \right) \]

Numerical analysis of this expression shows that it’s maximum is less then 1.67 (for \( a_0 = a_1 = b_0 = b_1 \approx 0.17 \)), hence \( I(\mathcal{X} : \Pi_B^k \mid \mathcal{Y}) + I(\mathcal{Y} : \Pi_A^k \mid \mathcal{X}) \leq 1.67. \)

6  Half-duplex communication with zero

As we have already mentioned before there are only two reasonable actions in this model: send 1 or receive. The following theorem shows that half-duplex communication with zero is more powerful than classical communication; namely, it is possible to compute equality in less than \( n \) rounds of communication.

\[ \textbf{Theorem 19.} D^{hd}_0(\text{EQ}_n) \leq \lfloor n / \log 3 \rfloor + 2 \lceil \log n \rceil + 1. \]

\[ \textbf{Proof.} \] Alice and Bob encode their inputs in ternary. In the first phase of the protocol, they process their inputs sequentially symbol by symbol in \( \lfloor n / \log 3 \rfloor \) rounds. At round \( i \) they process \( i \)th symbol in the following manner.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>receive</td>
<td>receive</td>
</tr>
<tr>
<td>1</td>
<td>send(1)</td>
<td>receive</td>
</tr>
<tr>
<td>2</td>
<td>receive</td>
<td>send(1)</td>
</tr>
</tbody>
</table>

In the next \( 2 \lceil \log n \rceil \) they send each other the number of ones they sent in the first phase. Depending on values of corresponding inputs, i.e., \( x_i \) and \( y_i \), we distinguish 6 types of witnesses of inequality: \((0, 1), (1, 0), (0, 2), (2, 0), (1, 2), (2, 1)\). If we make sure that each type can be detected by at least one of the players we are done. In the first phase, Alice can detect types \((0, 2), (2, 0), (2, 1)\), while Bob can detect types \((1, 0), (0, 1), \) and \((2, 1)\) (again). This leaves us with detecting witnesses of type \((1, 2)\). Assuming that there are no witnesses of other types, this will be detected in the second phase. \[ \text{\bf \small \&} \]

The best lower bound for this model is again for IP\(_n\). The next theorem is proved using round elimination (see full version [6] for the proof).

\[ \textbf{Theorem 20.} D^{hd}_0(\text{IP}_n) \geq n / \log \frac{2}{3 - \sqrt{5}} > n / 1.39. \]
The better lower is proved with information theoretic approach.

**Theorem 21.** \( D_a^{hd}(\text{IP}_n) \geq n/1.234. \)

**Proof.** The proof repeats the proof of Theorem 18. The only difference is that in this model players never send 0. So at the end we end up maximizing \((1 - b_1) \cdot h(a_1) + (1 - a_1) \cdot h(b_1), \) where \( h(p) = p \cdot \log \frac{1}{p} + (1 - p) \cdot \log \frac{1}{1 - p} \) is a binary entropy function. Maximum of this expression is slightly less then 1.234 \((a_1 = b_1 \approx 0.29). \)

## 7 Half-duplex communication with adversary

The main feature of this model is that receiving player cannot be 100% sure that the received bit if in fact is “real”, i.e., this bit originates from the other player, not from an adversary. The protocol must be correct for any strategy of the adversary. Our intuition prompts that in this setting silent and spent rounds would be useless. Using combinatorial methods, one can show the following two lower bounds (see full version [6] for the proof).

**Theorem 22.** \( D^{hd}_a(\text{EQ}_n) \geq n/\log 2.5. \)

**Theorem 23.** \( D^{hd}_a(\text{IP}_n) \geq n/\log 7. \)

And again better lower bound for IP\(_n\) can be obtained using information-theoretic approach.

**Theorem 24.** \( D^{hd}_a(\text{IP}_n) \geq n. \)

To prove this theorem we use the ideas from the proof of Theorem 18: in order to apply Lemma 12 we show that \( I(\mathcal{X} : \Pi^A_B | \mathcal{Y}) + I(\mathcal{Y} : \Pi^B_A | \mathcal{X}) \leq k, \) and hence we get the desired bound (see full version [6] for the detailed proof).

Using the same approach we can show \( 2 \log n \) lower bound on the complexity of Karchmer–Wigderson relation for parity function.

**Definition 25.** Let \( X = f^{-1}(0), Y = f^{-1}(1) \) for some Boolean function \( f : \{0, 1\}^n \rightarrow \{0, 1\}. \)

**The KW relation for function f,** \( R_f \subseteq X \times Y \times [n], \) is defined by \( R_f = \{(x, y, i) \mid x_i \neq y_i\}. \)

It well known that parity function \( \oplus_n : \{0, 1\}^n \rightarrow \{0, 1\}, \oplus_n(x) = \bigoplus_{i=1}^n x_i, \) requires \( n^2 \) formula size [9]. In the classical case it is equivalent to saying that KW relations for parity requires \( 2 \log n \) rounds of communication. In the proof of Theorem 24 we showed that \( I(\mathcal{X} : \epsilon^{k+1}_B | \mathcal{Y}, \Pi^B_A) + I(\mathcal{Y} : \epsilon^{k}_A | \mathcal{X}, \Pi^A_B) \leq 1. \) It allows us to prove the following analogue of this result.

**Corollary 26.** \( D^{hd}_a(R_{\oplus_n}) \geq 2 \log n. \)

**Proof.** Take the uniform distribution over valid input pairs with a single bit of difference. Then \( H(\mathcal{Y} | \mathcal{X}) + H(\mathcal{X} | \mathcal{Y}) = 2 \log n \) before any communication takes place. On the other hand it is easy to see that \( H(\mathcal{Y} | \mathcal{X}, \Pi A) + H(\mathcal{X} | \mathcal{Y}, \Pi B) = 0 \) at any leaf.

## 8 Open problems

The following table lists lower and upper bounds that we prove in this paper.

<table>
<thead>
<tr>
<th></th>
<th>EQ</th>
<th>IP</th>
<th>DISJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>( D^{hd}_a )</td>
<td>( \geq n/\log 5 )</td>
<td>( \geq n/1.67 )</td>
<td>( \leq n/2 + O(1) )</td>
</tr>
<tr>
<td>( D^{hd}_0 )</td>
<td>( \leq n/\log 5 + o(n) )</td>
<td>( \geq n/1.234 )</td>
<td></td>
</tr>
<tr>
<td>( D^{hd}_a )</td>
<td>( \geq n/\log 3 + o(n) )</td>
<td>( \geq n/\log 2.5 )</td>
<td>( \geq n )</td>
</tr>
</tbody>
</table>
It would be interesting to improve presented bounds to determine the true half-duplex complexity of these functions. We propose the following list of open problems.

1. Prove better upper and lower bounds for the half-duplex models with silence and zero.
2. Is there any $\alpha < 1$ such that for any $f : \{0, 1\}^n \times \{0, 1\}^n \to \{0, 1\}$, $D^{hd}_0(f) \leq \alpha n + o(n)$?
3. Is there any $f : \{0, 1\}^n \times \{0, 1\}^n \to \{0, 1\}$, such that at the same time $D(f) \geq n - o(n)$ and $D^{hd}_a(f) \leq \alpha n + o(n)$ for some $\alpha < 1$.

References

On the Complexity of Stable Fractional Hypergraph Matching

Takashi Ishizuka
Graduate School of Mathematics, Kyushu University
744 Motooka, Nishi-ku, Fukuoka 819-0395, Japan
ma218011@math.kyushu-u.ac.jp

Naoyuki Kamiyama
Institute of Mathematics for Industry, Kyushu University
JST, PRESTO
744 Motooka, Nishi-ku, Fukuoka 819-0395, Japan
kamiyama@imi.kyushu-u.ac.jp

Abstract
In this paper, we consider the complexity of the problem of finding a stable fractional matching in a hypergraphic preference system. Aharoni and Fleiner proved that there exists a stable fractional matching in every hypergraphic preference system. Furthermore, Kintali, Poplawski, Rajaraman, Sundaram, and Teng proved that the problem of finding a stable fractional matching in a hypergraphic preference system is \textit{PPAD}-complete. In this paper, we consider the complexity of the problem of finding a stable fractional matching in a hypergraphic preference system whose maximum degree is bounded by some constant. The proof by Kintali, Poplawski, Rajaraman, Sundaram, and Teng implies the \textit{PPAD}-completeness of the problem of finding a stable fractional matching in a hypergraphic preference system whose maximum degree is 5. In this paper, we prove that (i) this problem is \textit{PPAD}-complete even if the maximum degree is 3, and (ii) if the maximum degree is 2, then this problem can be solved in polynomial time. Furthermore, we prove that the problem of finding an approximate stable fractional matching in a hypergraphic preference system is \textit{PPAD}-complete.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases fractional hypergraph matching, stable matching, \textit{PPAD}-completeness

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.11

1 Introduction

The stable matching model introduced by Gale and Shapley [7] is one of the most important mathematical models for matching problems. The classical stable matching model is defined on undirected graphs. Thus, this model is naturally generalized to hypergraphs. It is not difficult to see that there exists an instance of the stable matching problem in hypergraphs that has no stable hypergraph matching. Thus, in this paper, we consider the following relaxation concept called a fractional matching. In the ordinary stable matching problem, the value 0 or 1 is assigned to each edge. On the other hand, in a fractional matching, a real number between 0 and 1 is assigned to each edge. Fortunately, it is known [1] that there exists a stable fractional matching in every hypergraph. The proof of this result in [1] was based on Scarf’s Lemma [16]. For example, the concept of stable fractional matchings in hypergraphs is used in [2, 3, 13]. It should be noted that stable fractional matchings

1 This research was supported by JST PRESTO Grant Number JPMJPR1753, Japan.
11:2 On the Complexity of Stable Fractional Hypergraph Matching

in hypergraphs are closely related to the stable matching problem with couples [4] that is a practically and theoretically important variant of the stable matching problem (see, e.g., [3, 13]). In this paper, we consider the problem of finding a stable fractional matching in a hypergraph.

For considering the computational complexity of a problem for which every instance is guaranteed to have a solution, Megiddo and Papadimitriou [11] introduced the complexity class TFNP that consists of all search problems in NP for which every instance is guaranteed to have a solution. The class PPAD introduced by Papadimitriou [14] is the class of all search problems such that the above property (i.e., every instance is guaranteed to have a solution) is proved by using a directed parity argument. Some problem A in PPAD is said to be PPAD-complete, if every problem in PPAD is reducible to A in polynomial time. The assumption that PPAD contains hard problems is considered as a reasonable hypothesis (see e.g., [15, Section 2.4.1]). Thus, it is reasonable to consider that a PPAD-complete problem is hard. For example, it is known [5, 6] that the problem of finding a Nash equilibrium [12] is PPAD-complete.

Kintali, Poplawski, Rajaraman, Sundaram, and Teng [10] proved that the problem of finding a stable fractional matching in a hypergraphic preference system is PPAD-complete. In this paper, we consider the complexity of the problem of finding a stable fractional matching in a hypergraphic preference system whose maximum degree is bounded by some constant. It is natural to consider that in many practical applications, the length of a preference list (i.e., the degree of a vertex) is constant. Thus, it is important to reveal the complexity of this problem with low constant degree. The proof by Kintali, Poplawski, Rajaraman, Sundaram, and Teng [10] implies the PPAD-completeness of the problem of finding a stable fractional matching in a hypergraphic preference system whose maximum degree is 5. However, to the best of our knowledge, the complexity of the problem of finding a stable fractional matching in a hypergraphic preference system whose maximum degree is at most 4 is open. In this paper, we prove that (i) this problem is PPAD-complete even if the maximum degree is 3, and (ii) if the maximum degree is 2, then this problem can be solved in polynomial time. Furthermore, we prove that the problem of finding an approximate stable fractional matching in a hypergraphic preference system is PPAD-complete.

2 Problem Formulation and Main Results

A hypergraphic preference system $P$ consists of the following two components. The first component is a finite hypergraph $(V, E)$. The second component is a set of strict total orders $\succ_v$ for vertices $v$ in $V$ such that for each vertex $v$ in $V$, $\succ_v$ is a strict total order on $E(v)$, where for each vertex $v$ in $V$, we denote by $E(v)$ the set of hyperedges $e$ in $E$ such that $v \in e$. We denote by $P = (V, E, \{\succ_v\})$ this hypergraphic preference system $P$. Notice that if $|e| = 2$ for every hyperedge $e$ in $E$, then $P$ is just an instance of the well-known stable roommate problem (see, e.g., [8]). Define $\deg(P) := \max_{v \in V} |E(v)|$.

Assume that we are given a hypergraphic preference system $P = (V, E, \{\succ_v\})$. Then a vector $x$ in $\mathbb{R}^E_+$ is called a fractional matching in $P$, if $\sum_{e \in E(v)} x(e) \leq 1$ for every vertex $v$ in $V$.

---

$^2$ A polynomial-time computable function $f$ is called a polynomial-time reduction from a problem $B$ in PPAD to a problem $A$ in PPAD, if for every instance $I_B$ of $B$, $f(I_B)$ is an instance of $A$, and furthermore there exists a polynomial-time computable function $g$ such that for every solution $y$ of $f(I_B)$, $g(y)$ is a solution of $I_B$. A problem $A$ in PPAD is said to be PPAD-complete, if for every problem $B$ in PPAD, there exists a polynomial-time reduction from $B$ to $A$. 
V, where $\mathbb{R}_+$ is the set of non-negative real numbers. Furthermore, a fractional matching $x$ in $\mathbb{R}_+^E$ is said to be stable, if for every hyperedge $e$ in $E$, there exists a vertex $v$ in $e$ such that

$$x(e) + \sum_{f \in E(v) : f > v \in e} x(f) = 1.$$  

It is known [1, Theorem 2.1] that there exists a stable fractional matching in every hypergraphic preference system. The problem called Fractional Hypergraph Matching is defined as follows. In this problem, we are given a hypergraphic preference system $P$. Then the goal of this problem is to find a stable fractional matching in $P$. The following result about the computational complexity of Fractional Hypergraph Matching is known.

| Theorem 1 (Kintali, Poplawski, Rajaraman, Sundaram, and Teng [10, Theorem 5.7]). Fractional Hypergraph Matching is PPAD-complete. |

The proof by Kintali, Poplawski, Rajaraman, Sundaram, and Teng [10] implies the PPAD-completeness of the problem of finding a stable fractional matching in a hypergraphic preference system $P$ such that $\deg(P) = 5$. However, to the best of our knowledge, the complexity of the problem of finding a stable fractional matching in a hypergraphic preference system $P$ such that $2 \leq \deg(P) \leq 4$ is open. (If $\deg(P) = 1$, then the answer of Fractional Hypergraph Matching is trivial.) In this paper, we prove the following theorems.

<table>
<thead>
<tr>
<th>Theorem 2. Fractional Hypergraph Matching in a hypergraphic preference system $P$ such that $\deg(P) = 3$ is PPAD-complete.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theorem 3. Fractional Hypergraph Matching in a hypergraphic preference system $P$ such that $\deg(P) = 2$ can be solved in polynomial time.</td>
</tr>
<tr>
<td>Theorem 4. Approximate Fractional Hypergraph Matching is PPAD-complete.</td>
</tr>
</tbody>
</table>

3 Proof of Theorem 2

For proving Theorem 2, we need the following lemma.

| Lemma 5. Assume that we are given a hypergraphic preference system $P$ such that $\deg(P) \geq 4$. Then there exists a hypergraphic preference system $Q$ such that (i) $\deg(Q) = 3$ and (ii) we can construct a stable fractional matching in $P$ from a stable fractional matching in $Q$ in polynomial time. Furthermore, we can construct $Q$ in polynomial time. |
Before proving Lemma 5, we prove Theorem 2 by using this lemma.

\textbf{Proof of Theorem 2.} It follows from Theorem 1 that \textsc{Fractional Hypergraph Matching} in a hypergraphic preference system $P$ such that $\deg(P) = 3$ is in \textsc{PPAD}. Furthermore, Theorem 1 and Lemma 5 imply that every problem in \textsc{PPAD} is reducible to \textsc{Fractional Hypergraph Matching} in a hypergraphic preference system $P$ such that $\deg(P) = 3$ in polynomial time. This completes the proof.

### 3.1 Proof of Lemma 5

In this subsection, we prove Lemma 5. The following proof is inspired by the proof of the \textsc{PPAD}-completeness of \textsc{Preference Game} with degree 3 by Kintali, Poplawski, Rajaraman, Sundaram, and Teng [10].

Assume that we are given a hypergraphic preference system $P = (V, E, \{\succ_v\})$ such that $\deg(P) \geq 4$. Then we construct a new hypergraphic preference system $Q = (W, F, \{\succ_v\})$ as follows (see Figure 1). Define

$$W := \{v_{i} | v \in V, i \in \{1, 2, \ldots, |E(v)|\}\} \cup \{v_{i} | v \in V, i \in \{1, 2, \ldots, |E(v)| - 1\}\}.$$ 

For each vertex $v$ in $V$ and each hyperedge $e$ in $E(v)$, we define

$$r(v, e) := 1 + \{|f \in E(v) | f \succ_v e\}.$$ 

For each hyperedge $e$ in $E$, we define $\tau := \{v_{r(v, e)} | v \in e\}$. Define $E := \{\tau | e \in E\}$ and

$$F := E \cup \{v_{i, \tau_i}, \{v_i, v_{i+1}\} | v \in V, i \in \{1, 2, \ldots, |E(v)| - 1\}\}.$$ 

For each vertex $v$ in $V$ and each integer $i$ in $\{1, 2, \ldots, |E(v)|\}$, we denote by $h^v_i$ the hyperedge $e$ in $E$ such that $v_i \in e$. For each vertex $w$ in $W$, we define the strict total order $\succ_w$ as follows. We first consider the case where $w = v_1$ for some vertex $v$ in $V$ and some integer $i$ in $\{1, 2, \ldots, |E(v)|\}$. It suffices to consider the case where $|E(v)| \geq 2$. In this case, we define

$$\begin{cases} 
h^v_1 \succ_w \{v_1, \tau_1\} & \text{if } i = 1 \\
\{v_{r(E(v)|-1, v_{E(v)}), v_{E(v)}}\} \succ_w h^v_{E(v)} & \text{if } i = |E(v)| \\
\{v_{i-1}, v_i\} \succ_w h^v_i & \text{otherwise.} \end{cases}$$

Next we assume that $w = \tau_i$ for some vertex $v$ in $V$ and some integer $i$ in $\{1, 2, \ldots, |E(v)| - 1\}$. In this case, we define $\{v_i, \tau_i\} \succ_w \{v_{i+1}, \tau_i\}$. Since $|W| \leq 2|V||E|$ and $|F| \leq |E| + 2|V||E|$, $Q$ can be constructed in polynomial time. Furthermore, $\deg(Q) = 3$.

In what follows, we prove that we can construct a stable fractional matching in $P$ from a stable fractional matching in $Q$ in polynomial time. Assume that we are given a stable fractional matching $z$ in $Q$. Then we define the vector $x$ in $\mathbb{R}^E_x$ by $x(e) := z(\tau)$. Clearly, we can construct $x$ from $z$ in polynomial time. What remains is to prove that $x$ is a stable fractional matching in $P$. For proving this, we need the following lemma.

\textbf{Lemma 6.} For every vertex $v$ in $V$ and every integer $i$ in $\{1, 2, \ldots, |E(v)| - 1\}$, \(z((v, \tau_i)) = 1 - \sum_{j=1}^{i} z(h^v_j)\), and

\(z((\tau_i, v_{i+1})) = \sum_{j=1}^{i} z(h^v_j)\).
The copies of $v$ in $Q$ and the hyperedges containing these copies. For every integer $i$ in $\{1,2,3,4\}$, we have $h_i = \overline{v}$. 

**Proof.** Let $v$ be a vertex in $V$ such that $|E(v)| \geq 2$. We prove by induction on $i$.

We first consider the case of $i = 1$. Since $z$ is a fractional matching in $Q$, we have

$$1 \geq \sum_{e \in F(v_1)} z(e) = z(h_i^v) + z(\{v_1, \overline{v}_1\}).$$

This implies that $z(\{v_1, \overline{v}_1\}) \leq 1 - z(h_i^v)$. For proving (T1) by contradiction, we assume that $z(\{v_1, \overline{v}_1\}) < 1 - z(h_i^v)$. Since $z$ is a stable fractional matching in $Q$, at least one of the following statements holds.

$$1 = z(\{v_1, \overline{v}_1\}) + \sum_{e \in F(v_1) : e \not\supset v_1} z(e) = z(\{v_1, \overline{v}_1\}) + z(h_i^v), \quad (1)$$

$$1 = z(\{v_1, \overline{v}_1\}) + \sum_{e \in F(\overline{v}_1) : e \not\supset \overline{v}_1} z(e) = z(\{v_1, \overline{v}_1\}). \quad (2)$$

However, since $z(h_i^v) \geq 0$, the above assumption implies that $z(\{v_1, \overline{v}_1\}) + z(h_i^v) < 1$ and $z(\{v_1, \overline{v}_1\}) < 1$. These observations contradict (1) and (2). Thus, $z(\{v_1, \overline{v}_1\}) = 1 - z(h_i^v)$.

Next we consider (T2). Since $z$ is a fractional matching in $Q$, we have

$$1 \geq \sum_{e \in F(\overline{v}_1)} z(e) = z(\{v_1, \overline{v}_1\}) + z(\{\overline{v}_1, v_2\}).$$

Since (T1) for the case of $i = 1$ implies that $z(\{v_1, \overline{v}_1\}) = 1 - z(h_i^v)$, we have $z(\{\overline{v}_1, v_2\}) \leq z(h_i^v)$. For proving (T2) by contradiction, we assume that $z(\{\overline{v}_1, v_2\}) < z(h_i^v)$. Since $z$ is a stable fractional matching in $Q$, at least one of the following statements holds.

$$1 = z(\{\overline{v}_1, v_2\}) + \sum_{e \in F(\overline{v}_1) : e \not\supset \overline{v}_1} z(e) = z(\{\overline{v}_1, v_2\}) + z(\{v_1, \overline{v}_1\}). \quad (3)$$

$$1 = z(\{\overline{v}_1, v_2\}) + \sum_{e \in F(v_2) : e \not\supset v_2} z(e) = z(\{\overline{v}_1, v_2\}). \quad (4)$$

Since (T1) for the case of $i = 1$ implies that $z(\{v_1, \overline{v}_1\}) = 1 - z(h_i^v)$, the above assumption implies that

$$z(\{\overline{v}_1, v_2\}) + z(\{v_1, \overline{v}_1\}) = z(\{\overline{v}_1, v_2\}) + 1 - z(h_i^v) < z(h_i^v) + 1 - z(h_i^v) = 1.$$
This contradicts (3). Furthermore, since $z$ is a fractional matching in $Q$, we have $z(h_1^v) \leq 1$. Thus, the above assumption implies that $z([v_1, v_2]) < 1$. This contradicts (4), and completes the proof of $z([\overline{v}_1, v_2]) = z(h_1^v)$.

Let $k$ be an integer in $\{2, 3, \ldots, |E(v)| - 1\}$, and we assume that this lemma holds in the case of $i = k - 1$. Then we prove that this lemma holds in the case of $i = k$. Since $z$ is a fractional matching in $Q$, we have

$$1 \geq \sum_{e \in F(v_k)} z(e) = z([\overline{v}_{k-1}, v_k]) + z(h_k^v) + z([v_k, \overline{v}_k]).$$

Since the induction hypothesis implies that

$$z([\overline{v}_{k-1}, v_k]) + z(h_k^v) + z([v_k, \overline{v}_k]) = \sum_{j=1}^{k-1} z(h_j^v) + z([v_k, \overline{v}_k]),$$

we have

$$z([v_k, \overline{v}_k]) \leq 1 - \sum_{j=1}^{k-1} z(h_j^v). \quad (5)$$

For proving (T1) by contradiction, we assume that the inequality in (5) strictly holds. Since $z$ is a stable fractional matching in $Q$, at least one of the following statements holds.

$$1 = z([v_k, \overline{v}_k]) + \sum_{e \in F(v_k) \setminus v_k, \overline{v}_k} z(e) = z([v_k, \overline{v}_k]) + z([\overline{v}_{k-1}, v_k]) + z(h_k^v). \quad (6)$$

$$1 = z([v_k, \overline{v}_k]) + \sum_{e \in F(v_k) \setminus v_k, \overline{v}_k} z(e) = z([v_k, \overline{v}_k]). \quad (7)$$

However, the above assumption and the induction hypothesis imply that

$$z([v_k, \overline{v}_k]) + z([\overline{v}_{k-1}, v_k]) + z(h_k^v) = z([v_k, \overline{v}_k]) + \sum_{j=1}^{k-1} z(h_j^v) + z(h_k^v)$$

$$< 1 - \sum_{j=1}^{k-1} z(h_j^v) + \sum_{j=1}^{k} z(h_j^v) = 1.$$

This contradicts (6). Furthermore, since $z \in \mathbb{R}_+^F$, $z([v_k, \overline{v}_k]) < 1$ follows from the above assumption. This contradicts (7). This completes the proof of (T1).

Next we consider (T2). Since $z$ is a fractional matching in $Q$, we have

$$1 \geq \sum_{e \in F(v_k)} z(e) = z([v_k, \overline{v}_k]) + z([\overline{v}_k, v_{k+1}]).$$

Since (T1) for the case of $i = k$ implies that

$$z([v_k, \overline{v}_k]) = 1 - \sum_{j=1}^{k} z(h_j^v), \quad (8)$$

we have

$$z([\overline{v}_k, v_{k+1}]) \leq \sum_{k=1}^{k} z(h_j^v). \quad (9)$$
For proving (T2) by contradiction, we assume that the inequality in (9) strictly holds. Since $z$ is a stable fractional matching in $Q$, at least one of the following statements holds.

$$1 = z(\{v_k, v_{k+1}\}) + \sum_{e \in P(\overline{v}_k) \setminus \overline{R}_k} z(e) = z(\{v_k, v_{k+1}\}) + z(\{v_k, \overline{v}_k\}). \tag{10}$$

$$1 = z(\{v_k, v_{k+1}\}) + \sum_{e \in P(v_{k+1}) \setminus \overline{R}_k} z(e) = z(\{v_k, v_{k+1}\}). \tag{11}$$

Notice that (8) and the above assumption implies that

$$z(\{v_k, v_{k+1}\}) + z(\{v_k, \overline{v}_k\}) < \sum_{j=1}^{k} z(h_j^v) + 1 - \sum_{j=1}^{k} z(h_j^v) = 1.$$

This contradicts (10). Furthermore, (8) and $z \in \mathbb{R}_+^F$ imply that $\sum_{j=1}^{k} z(h_j^v) \leq 1$. This and the above assumption imply that $z(\{v_k, v_{k+1}\}) < 1$. This contradicts (11), and completes the proof. □

We are now ready to prove that $x$ is a stable fractional matching in $P$. Let $v$ be a vertex in $V$. Define $k := |E(v)|$. If $k = 1$, then

$$\sum_{e \in E(v)} x(e) = z(h_1^v) \leq 1.$$

If $k > 1$, then

$$\sum_{e \in E(v)} x(e) = \sum_{i=1}^{k} z(h_i^v) = \sum_{i=1}^{k-1} z(h_i^v) + z(h_k^v) = z(\{v_{k-1}, v_k\}) + z(h_k^v) \quad \text{(by (T2) of Lemma 6)}$$

$$= \sum_{e \in E(v_k)} z(e) \leq 1,$$

where the inequality follows from the fact that $z$ is a fractional matching in $Q$.

Lastly, we prove that $x$ is a stable fractional matching in $P$. Let $e$ be a hyperedge in $E$. Then since $z$ is a stable fractional matching in $Q$, there exists a vertex $w$ in $\overline{v}$ such that

$$z(\overline{v}) + \sum_{f \in F(w) \setminus \overline{v}} z(f) = 1.$$

Assume that $w = v_k$ for some vertex $v$ in $e$ and some integer $k$ in $\{1, 2, \ldots, |E(v)|\}$. Notice that $\overline{v} = h_k^v$. For each integer $i$ in $\{1, 2, \ldots, k\}$, we assume that $h_i^v = \overline{v}_i$. Notice that $e_k = e$, $e_1 \succ v_2 \succ v_3 \succ \cdots \succ v_{k-1}$, and $e \succ v_f$ holds for every hyperedge $f$ in $E(v) \setminus \{e_1, e_2, \ldots, e_k\}$. For each integer $i$ in $\{1, 2, \ldots, k\}$, $x(e_i) = z(h_i^v)$. If $k = 1$, then

$$1 = z(\overline{v}) + \sum_{f \in F(w) \setminus \overline{v}} z(f) = z(\overline{v}) = x(e) + \sum_{f \in E(v) \setminus \overline{v}} x(f).$$

\begin{document}
\end{document}
On the Complexity of Stable Fractional Hypergraph Matching

![Hypergraph and Directed Graph](image)

**Figure 2** (a) A hypergraph $H = (V, E)$ such that $e_3 \succ v_1, e_2 \succ v_2, e_1 \succ v_3, e_4 \succ v_4, e_5 \succ v_4, e_6 \succ v_6, e_7 \succ v_8$, and $e_7 \succ v_9$. (b) The directed graph $D$ constructed from $H$.

If $k > 1$, then

$$1 = z(\pi) + \sum_{f \in F(v) : f \succ_v \pi} z(f)$$

$$= z(h_k^v) + z(\{\pi_{k-1}, v_k\})$$

$$= z(h_k^v) + \sum_{i=1}^{k-1} z(h_i^v) \quad \text{(by (T2) of Lemma 6)}$$

$$= x(e) + \sum_{i=1}^{k-1} x(e_i)$$

$$= x(e) + \sum_{f \in E(v) : f \succ_v e} x(f).$$

These imply that $x$ is a stable fractional matching in $P$. This completes the proof.

## 4 Proof of Theorem 3

Throughout this section, we assume that we are given a hypergraphic preference system $P$ such that $\deg(P) = 2$. Define $V^*$ as the set of vertices $v$ in $V$ such that $|E(v)| = 2$. In addition, we define the directed graph $D = (N, A)$ as follows. For each hyperedge $e$ in $E$, $N$ contains a vertex $n_e$. For each vertex $v$ in $V^*$, $A$ contains an arc from $n_f$ to $n_e$, where we assume that distinct hyperedges $e, f$ in $E$ contain $v$ and $e \succ_v f$. See Figure 2 for an example of $D$.

Our algorithm is described in Algorithm 1. This algorithm can be intuitively explained as follows. If there exists a vertex $n_e$ in $N$ such that any arc in $A$ does not leave this vertex, we set the value of $e$ is most preferred by every vertex $v$ in $e$. Thus, we set the value for $e$ to be 1. For every arc $a = (n_f, n_e)$ in $A$, since some vertex in $V$ is contained in $e, f$, we must set the value of $f$ to be 0. Then we can remove vertices in $N$ whose value is determined from $D$. We repeat this. Finally, we obtain a directed graph $D'$ in which the out-degree of every vertex is at least one. Thus, by setting the value for each vertex of $D'$ to be $1/2$, we can construct a stable fractional matching in $P$.

Here we apply Algorithm 1 for the example in Figure 2. Since $n_{e_3}$ is the only vertex such that any arc in $A$ does not leave this vertex, we set $\xi_2(n_{e_3}) := 1$ and the value of $\xi_2$ for other vertex is equal to 0. Then the vertices $n_{e_3}, n_{e_6}, n_{e_7}$ (and the arcs around them) are...
Algorithm 1:

1. Define $D_1 := D$ and $N_1 := N$.
2. Define the vector $\xi_1$ in $\mathbb{R}^N_+$ by $\xi_1(v) := 0$ for each vertex $v$ in $N$.
4. while there exists a vertex $v$ in $N_t$ such that any arc of $D_t$ does not leave $v$ do
   5. Define $S_t$ as the set of vertices $v$ in $N_t$ such that any arc of $D_t$ does not leave $v$.
   6. Define the vector $\xi_{t+1}$ in $\mathbb{R}^N_+$ by $\xi_{t+1}(v) := 1$ for each vertex $v$ in $S_t$ and $\xi_{t+1}(v) := \xi_t(v)$ for each vertex $v$ in $N \setminus S_t$.
   7. Define $T_t$ as the set of vertices $v$ in $N_t$ such that there exists an arc of $D_t$ from $v$ to some vertex in $S_t$.
   8. Define $N_{t+1} := N_t \setminus (S_t \cup T_t)$, and $D_{t+1}$ as the subgraph of $D_t$ induced by $N_{t+1}$.
   9. Set $t := t + 1$.
10. end
11. Define the vector $\xi^*$ in $\mathbb{R}^N_+$ by $\xi^*(v) := 1/2$ for each vertex $v$ in $N_t$ and $\xi^*(v) := \xi_t(v)$ for each vertex $v$ in $N \setminus N_t$.
12. Define the vector $x$ in $\mathbb{R}^E_+$ by $x(e) := \xi^*(e)$ for each hyperedge $e$ in $E$.
13. Output $x$, and halt.

removed. In the remaining graph, for every vertex, at least one arc leaves it. Thus, the value 1/2 are assigned to the remaining vertices, and the algorithm halts. In the obtained stable fractional matching $x$, $x(e_i) = 1/2$ for every integer $i$ in $\{1, 2, 3, 4\}$, $x(e_5) = 0$, $x(e_6) = 0$, and $x(e_7) = 1$.

Lemma 7. The output of Algorithm 1 is a stable fractional matching in $P$.

Proof. Assume that Algorithm 1 halts when $t = k$. For proving this lemma, it suffices to prove the following conditions are satisfied.

(P1) For every arc $a = (u, v)$ in $A$, we have $\xi^*(u) + \xi^*(v) \leq 1$.
(P2) For every vertex $v$ in $N$ such that $\xi^*(v) \neq 1$, there exist a vertex $w$ in $N$ such that an arc from $v$ to $w$ is contained in $A$ and $\xi^*(v) + \xi^*(w) = 1$.

We first prove (P1). Assume that we are given an arc $a = (u, v)$ in $A$. If $\xi^*(u) = 0$, then (P1) clearly holds. Next we assume that $\xi^*(u) = 1$. Then there exists a positive integer $t$ such that $u \in N_t$ and any arc of $D_t$ does not leave $u$. Notice that $v \notin N_t$. This implies that $\xi^*(v) \in \{0, 1\}$. If $\xi^*(v) = 1$, then then there exists a positive integer $t'$ such that $t' < t$, $v \in N_{t'}$, and any arc of $D_t$ does not leave $v$. Furthermore, the definition of $T_t$ implies that $u \in T_t$. This implies that $u \notin N_t$, which contradicts the fact that $u \in N_t$. Thus, we have $\xi^*(v) = 0$. Lastly, we consider the case where $\xi^*(u) = 1/2$, i.e., $u \in N_k$. If $\xi^*(v) = 1$, then $u \notin N_k$, which contradicts the fact that $u \in N_k$. This implies that $\xi^*(v) \in \{0, 1/2\}$. This completes the proof of (P1).

Next we prove (P2). Assume that we are given a vertex $v$ in $N$ such that $\xi^*(v) \neq 1$. Assume that $\xi^*(v) = 0$. In this case, there exists a positive integer $t$ such that $v \in T_t$. That is, there exists a vertex $w$ in $S_t$ such that there exists an arc of $D_t$ from $v$ to $w$. Since $w \in S_t$, $\xi^*(w) = 1$. This implies that $\xi^*(v) + \xi^*(w) = 1$. Next we assume that $\xi^*(v) = 1/2$. In this case, there exists a vertex $w$ in $N_k$ such that there exists an arc of $D_k$ from $v$ to $w$. Since $\xi^*(w) = 1/2$, we have $\xi^*(v) + \xi^*(w) = 1$. This completes the proof.

Proof of Theorem 3. This theorem immediately follows from Lemma 7.
5 Proof of Theorem 4

In this section, we prove Theorem 4. Since a stable fractional matching is clearly an \( \epsilon \)-stable fractional matching for any positive rational number \( \epsilon \), Theorem 1 (i.e., the fact that \text{Fractional Hypergraph Matching} is in \text{PPAD}) implies that \text{Approximate Fractional Hypergraph Matching} is in \text{PPAD}. What remains is to prove that every problem in \text{PPAD} is reducible to \text{Approximate Fractional Hypergraph Matching} in polynomial time. For this, Theorem 1 implies that it is sufficient to prove that \text{Fractional Hypergraph Matching} is reducible to \text{Approximate Fractional Hypergraph Matching} in polynomial time. This fact immediately follows from the following lemma.

\begin{lemma}
Assume that we are given a hypergraphic preference system \( P = (V, E, \{ \succ_v \}) \). Furthermore, we define \( \epsilon := 1/2^{20|E|^4} \). Then we can construct a stable fractional matching in \( P \) from an \( \epsilon \)-stable fractional matching in \( P \) in polynomial time.
\end{lemma}

What remains is to prove Lemma 8. We prove Lemma 8 by using the following known result called LP compactness. Assume that we are given positive integers \( m, n \) and vectors \( a \) in \( \mathbb{Q}^{m \times n} \) and \( b \) in \( \mathbb{Q}^m \), where \( \mathbb{Q} \) is the set of rational numbers. Then we consider the following linear inequality system whose variable is a vector \( x \) in \( \mathbb{R}^n \).

\[
\sum_{j=1}^n a(i,j) \cdot x(j) \geq b(i) \quad (i \in \{1, 2, \ldots, m\}).
\]  

(12)

For each positive real number \( \delta \) and each vector \( y \) in \( \mathbb{R}^n \), we say that \( y \) satisfies the linear inequality system (12) to within \( \delta \), if

\[
\sum_{j=1}^n a(i,j) \cdot y(j) \geq b(i) - \delta
\]

for every integer \( i \) in \( \{1, 2, \ldots, m\} \).

\begin{theorem}[LP compactness (see [10, Lemma 4.11])] Assume that we are given positive integers \( m, n \) and vectors \( a \) in \( \mathbb{Q}^{m \times n} \) and \( b \) in \( \mathbb{Q}^m \). Furthermore, we assume that there exists a positive integer \( \beta \) satisfying the condition that for every pair of integers \( i \) in \( \{1, 2, \ldots, m\} \) and \( j \) in \( \{1, 2, \ldots, n\} \), there exist integers \( p, q, r, s \) such that \( a(i,j) = p/q, b(i) = r/s \), and \( |p|, |q|, |r|, |s| \leq 2^\beta \). Then we consider the following linear inequality system whose variable is a vector \( x \) in \( \mathbb{R}^n \).

\[
\sum_{j=1}^n a(i,j) \cdot x(j) \geq b(i) \quad (i \in \{1, 2, \ldots, m\}).
\]  

(13)

If there exists a vector \( y \) in \( \mathbb{R}^n \) satisfying the linear inequality system (13) to within \( 1/2^{20m^4\beta} \), then there exists a vector \( x \) in \( \mathbb{R}^n \) that is feasible for the linear inequality system (13).

We are now ready to prove Lemma 8.

Proof of Lemma 8. Assume that we are given an \( \epsilon \)-stable fractional matching \( y \) in \( P \). For each hyperedge \( e \) in \( E \), we define set \( U(e) \) as the set of vertices \( v \) in \( e \) such that

\[
y(e) + \sum_{f \in U(e): f \succ e} y(f) \geq 1 - \epsilon.
\]
Notice that since $y$ in an $\epsilon$-stable fractional matching in $P$, $U(e) \neq \emptyset$ for any hyperedge $e$ in $E$. We consider the following linear inequality system whose variable is a vector $x$ in $\mathbb{R}^E$.

\[- \sum_{e \in E(v)} x(e) \geq -1 \quad (v \in V)\]

\[x(e) + \sum_{f \in E(v): f \succ v e} x(f) \geq 1 \quad (e \in E, v \in U(e))\]

\[x(e) \geq 0 \quad (e \in E).\] (14)

Notice that the number of constraints of the linear inequality system (14) is bounded by a polynomial in the input size of Fractional Hypergraph Matching.

Notice that $y$ satisfies the linear inequality system (14) to within $1/2^{20|E|^4}$. Thus, by setting $n := |E|$ and $\beta := 1$, Theorem 9 implies that there exists a vector $x$ in $\mathbb{R}^E$ that is feasible for the linear inequality system (14). Notice that we can find a vector $x$ in $\mathbb{R}^E$ that is feasible for the linear inequality system (14) in polynomial time by using the ellipsoid method [9].

Let $x$ be a vector in $\mathbb{R}^E$ that is feasible for the linear inequality system (14). Then we prove that $x$ is a stable fractional matching in $P$. For this, it suffices to prove that for every hyperedge $e$ in $E$, there exists a vertex $v$ in $e$ such that

\[x(e) + \sum_{f \in E(v): f \succ v e} x(f) = 1.\] (15)

Let $e$ be a hyperedge in $E$. The first constraint of (14) implies that

\[x(e) + \sum_{f \in E(v): f \succ v e} x(f) \leq 1\]

for every vertex $v$ in $U(e)$. Thus, the second constraint of (14) implies that

\[x(e) + \sum_{f \in E(v): f \succ v e} x(f) = 1\]

for every vertex $v$ in $U(e)$. Since $U(e) \neq \emptyset$, this implies that there exists a vertex $v$ in $e$ satisfying (15). This completes the proof. ▶

References

On the Complexity of Stable Fractional Hypergraph Matching


Deciding the Closure of Inconsistent Rooted Triples Is NP-Complete

Matthew P. Johnson
Department of Computer Science, Lehman College
Ph.D. Program in Computer Science, The Graduate Center
City University of New York, USA

Abstract
Interpreting three-leaf binary trees or rooted triples as constraints yields an entailment relation, whereby binary trees satisfying some rooted triples must also thus satisfy others, and hence a closure operator, which is known to be polynomial-time computable. This is extended to inconsistent triple sets by defining that a triple is entailed by such a set if it is entailed by any consistent subset of it.

Determining whether the closure of an inconsistent rooted triple set can be computed in polynomial time was posed as an open problem in the Isaac Newton Institute’s “Phylogenetics” program in 2007. It appears (as NC4) in a collection of such open problems maintained by Mike Steel, and it is the last of that collection’s five problems concerning computational complexity to have remained open. We resolve the complexity of computing this closure, proving that its decision version is NP-Complete.

In the process, we also prove that detecting the existence of any acyclic B-hyperpath (from specified source to destination) is NP-Complete, in a significantly narrower special case than the version whose minimization problem was recently proven NP-hard by Ritz et al. This implies it is NP-hard to approximate (our special case of) their minimization problem to within any factor.

2012 ACM Subject Classification Mathematics of computing → Trees, Mathematics of computing → Hypergraphs, Theory of computation → Problems, reductions and completeness, Applied computing → Molecular evolution

Keywords and phrases phylogenetic trees, rooted triple entailment, NP-Completeness, directed hypergraphs, acyclic induced subgraphs, computational complexity


Acknowledgements This work was supported in part by NSF award INSPIRE-1547205, and by the Sloan Foundation via a CUNY Junior Faculty Research Award.

1 Introduction

We investigate the computational complexity of a problem in which, based on a given collection of relationships holding between the leaves of a hypothetical (rooted) binary tree $T$, the task is to infer whatever additional relationships (of the same form) must also hold between $T$’s leaves as a consequence. Various problems in phylogenetic tree reconstruction involve inference of this kind. The specific relationship form in question here, obtaining between some three leaves $p, q, o$ and denoted $pq|o$, is that of the path between $p$ and $q$ being node-disjoint from the path between $o$ and the root, or equivalently, of the lowest common ancestor (lca) of $p$ and $q$ not being an ancestor of $o$. This relationship is modeled as a rooted triple, i.e., the (rooted, full) binary tree on leaves $p, q, o$ in which $p$ and $q$ are siblings, and their parent and $o$ are both children of the root. Then $pq|o$ holding in $T$ is equivalent to having the subtree of $T$ induced by $p, q, o$ be homeomorphic to $pq|o$’s corresponding three-leaf binary tree.
Deciding the Closure of Inconsistent Rooted Triples Is NP-Complete

The problem of computing the set of all rooted triples entailed by a given triple set $R'$ (its closure $\overline{R'}$) is known to be polynomial-time computable by, e.g., Aho et al.'s BUILD algorithm [6, 1] if $R'$ is consistent, i.e., if there exists a binary tree satisfying all triples in $R'$. 

If a rooted triple set $R$ is inconsistent, then a given triple is said to be entailed by $R$ if it is entailed by any consistent subset $R' \subset R$. That is, the closure $\overline{R}$ equals the union of the closures of all $R'$'s consistent subsets. Thus the naive brute-force algorithm for computing $\overline{R}$ suggested by the definition is exponential-time in $|R|$. 

Determining the complexity of the problem of computing $\overline{R}$ was posed in the Isaac Newton Institute's "Phylogenetics" program in 2007 [9], and it appears (as NC4) in a collection of such open problems maintained by Mike Steel [13]. That collection's other four problems concerning computational complexity were all solved by 2009 or 2010, but NC4 has remained open. We resolve the complexity of computing $\overline{R}$, proving that it is NP-hard. In particular, we prove that its decision version, i.e., deciding whether a given rooted triple is entailed by $R$, is NP-Complete. 

In the process, we also obtain stronger hardness results for a problem concerning acyclic B-hyperpaths, a directed hypergraph problem that has recently been applied to another computational biology application, but interestingly one unrelated to phylogenetic trees and rooted triples: signaling pathways, the sequences of chemical reactions through which cells respond to signals from their environment (see Ritz et al. [11]). 

Specifically, we prove that detecting the existence of any acyclic B-hyperpath (between specified source and destination) is NP-Complete, in a significantly narrower special case (viz., the case in which every hyperarc has one tail and two heads) than the version whose minimization problem was recently proven NP-hard by Ritz et al. This immediately implies it is NP-hard to approximate (our special case of) their minimization problem to within any factor. Moreover, even if we restrict ourselves to feasible problem instances (i.e., those for which there exists at least one such acyclic B-hyperpath), we show that this “promise problem” [8] special case is NP-hard to approximate to within factor $|V|^{1-\epsilon}$ for all $\epsilon > 0$. 

Related Work

Inference of new triples from a given set of rooted triples holding in a binary tree was studied by Bryant and Steel [6, 5], who proved many results on problems involving rooted triples, as well as quartets, and defined the closure of an inconsistent triple set. The polynomial-time BUILD algorithm of Aho et al. [1] (as well as subsequent extensions and speedups) can be used to construct a tree satisfying all triples in $R$ (and to obtain the closure $\overline{R}$), or else to conclude than none exists. 

Gallo et al. [7] defined a number of basic concepts involving paths and cycles in directed hypergraphs, including B-connectivity. Ausiello et al. [2] studied path and cycle problems algorithmically in directed hypergraphs and showed, via a simple reduction from SET COVER, that deciding whether there exists a B-hyperpath from specified source to destination with $\leq \ell$ hyperarcs is NP-Complete. Ritz et al. [11] recently studied a problem involving “signaling hypergraphs”, which are directed hypergraphs that can contain “hypernodes”. They modify Ausiello et al.’s hardness reduction from SET COVER to show that deciding the existence of a length $\leq \ell$ B-hyperpath is NP-Complete already in the special case of directed hypergraphs each of whose hyperarcs has at most 3 head nodes and at most 3 tail nodes (due to SET COVER becoming hard once sets of size 3 are permitted). Ritz et al.’s hardness proof actually does not use the fact that their problem formulation requires the computed B-hyperpath to be acyclic. Because the entire directed hypergraph they construct is (like Ausiello et al.’s) always acyclic, their proof provides hardness regardless of whether the formulation includes...
an acyclicity constraint. This constraint is essential to our hardness proof, however, so our result does not rule out the possibility that a B-hyperpath minimization problem formulation without an acyclicity requirement would be easier to approximate.

2 Preliminaries

2.1 Rooted Triples

▶ Definition 1. For any nodes u, v of a rooted binary tree (or simply a tree):
- v ≤ u denotes that v is a descendant of u (and u is an ancestor of v), i.e., u appears on the path from v to the root; v < u denotes that v is a proper descendant of u (and u is a proper ancestor of v), i.e., v ≤ u and v ≠ u.
- uv denotes their lowest common ancestor (lca), i.e., the node w of maximum distance from the root that satisfies w ≥ u and w ≥ v.

▶ Definition 2.
- A rooted triple (or simply a triple) t = (\{p, q, o\}, o) ∈ (\frac{1}{2}) × L (with p, q, o all distinct, for an underlying finite leaf set L) is denoted by the shorthand notation pq|o and represents the constraint: the path from p to q is node-disjoint from the path from o to the root.
- The left-hand side (LHS) of a triple pq|o is pq, and its right-hand side (RHS) is o.
- L(T) denotes the set of leaves of a tree T, and L(R') denotes the set of leaves appearing in any of the triples within a set R', i.e., L(R') = \bigcup_{p,q,o \in R'} \{p,q,o\}.
- A tree T with p, q, o ∈ L(T) displays the triple pq|o (or, pq|o holds in T) if the corresponding constraint holds in T. The set of all trees displayed by T is denoted by r(T). The set of all trees that display all triples in R' is denoted by (R'). A set of triples R' is consistent if (R') is nonempty.

▶ Definition 3.
- For a consistent triple set R', a given triple t (which may or may not be a member of R') is entailed by R', denoted R' ⊢ t, if every tree displaying all the triples in R' also displays t, i.e., if t is displayed by every tree in (R'). The closure \overline{R} is the set of all triples entailed by R', i.e., \overline{R} = \{t : R' ⊢ t\}, which can also be defined as \overline{R} = \bigcap_{T \in \langle R' \rangle} r(T) [6].
- For an inconsistent triple set R, a given triple t (which may or may not be a member of R) is entailed by R, again denoted R ⊢ t, if there exists a consistent subset R' ⊂ R that entails t. The closure \overline{R} is again the set of all triples entailed by R, or equivalently the union, taken over every consistent subset R' ⊂ R, of \overline{R}, i.e., \bigcup_{\text{cons. } R' \subset R} \overline{R}.

We first state a few immediate consequences of these definitions.

▶ Observation 4.
- It can happen that pp' = qq' even if \{p, p'\} ∩ \{q, q'\} = ∅.
- In any given tree T having p, q, o ∈ L(T), exactly one of pq|o, po|q, and qo|p holds.
- pq|o if and only if (path: p to q) ∩ (path: o to the root) = ∅ if and only if pq < po = qo.
- Equivalently, the 3-point condition for ultrametrics [12] holds: for all p, q, o ∈ L(T), we have pq < po = qo or qo < po = qp or OP < qo = pq.
- Regardless of whether triple set R is consistent, its closure \overline{R} satisfies R ⊂ \overline{R} ⊆ (\frac{1}{2}) × L, and so |\overline{R}| = O(|L|^3).

We state the problem formally.
12:4 Deciding the Closure of Inconsistent Rooted Triples Is NP-Complete

Table 1 Variable name conventions, many of which (also) represent leaves in the triple set $R$ constructed in the reduction. Note that the notation $pq$ (for leaves $p, q$) is used to denote both $\text{lca}(p, q)$ and the hypergraph node whose outgoing hyperarcs represent triples of the form $pq|o$, i.e., those constraining $\text{lca}(p, q)$ from above.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p, q, p', q', o, o'$</td>
<td>generic leaf variables, especially in triples’ LHSs or RHSs (resp.) (leaves)</td>
</tr>
<tr>
<td>$b_i, b'_i, c_i, d_i$, etc.</td>
<td>particular leaf names (leaves)</td>
</tr>
<tr>
<td>$pq$, etc.</td>
<td>lowest common ancestor lca($p, q$) of leaves $p, q$ (leaf 2-sets)</td>
</tr>
<tr>
<td>$\alpha, \beta, \gamma$</td>
<td>leaves of target triple $\alpha\beta\gamma$ (leaves)</td>
</tr>
<tr>
<td>$t$</td>
<td>rooted triple, especially of form $p_kq_k</td>
</tr>
<tr>
<td>$R$ or $R'$</td>
<td>set of triples, especially inconsistent or consistent (resp.) (leaf sets)</td>
</tr>
<tr>
<td>$L$ or $L(R)$</td>
<td>set of leaves or set of leaves appearing in members of $R$ (resp.) (leaf sets)</td>
</tr>
<tr>
<td>$u, u_k, v, v', v_k, v'_k$</td>
<td>hypergraph nodes, especially tail node or head nodes (resp.) (leaf 2-sets)</td>
</tr>
<tr>
<td>$pq$, etc.</td>
<td>hypergraph node corresponding to leaves $p, q$ (leaf 2-sets)</td>
</tr>
<tr>
<td>$\alpha\beta, \alpha\beta_1, \alpha\beta_2, \alpha\beta_3$</td>
<td>source and destination nodes (resp.) (leaf 2-sets)</td>
</tr>
<tr>
<td>$a_k$</td>
<td>1-2-hyperarc, especially of form $u_k \rightarrow {v_k, v'_k} = p_kq_k \rightarrow {p_ko_k, q_ko_k}$, with $k \in [\ell] = {1, ..., \ell}$ indicating $a_k$’s position in a path $P$ of length $</td>
</tr>
<tr>
<td>$x_i$</td>
<td>$i$th SAT variable, with $i \in [n]$ (leaves)</td>
</tr>
<tr>
<td>$C_j$</td>
<td>$j$th SAT clause, with $j \in [m]$ (leaves)</td>
</tr>
<tr>
<td>$x_i, \bar{x}_i$ or $\bar{x}_i$</td>
<td>literals (positive, negative or either, resp.) of $x_i$ (leaves)</td>
</tr>
<tr>
<td>$x^j_i, \bar{x}^j_i$ or $\bar{x}^j_i$</td>
<td>the appearance (positive, negative or either, resp.) of $x_i$ in $C_j$ (leaves)</td>
</tr>
<tr>
<td>$x^j_i, \bar{x}^j_i$ or $\bar{x}^j_i$</td>
<td>the $w$th variable appearance in $C_j$ (leaves)</td>
</tr>
<tr>
<td>$x^j, \bar{x}^j$ or $\bar{x}^j$</td>
<td>some (unspecified) variable appearance in $C_j$ (leaves)</td>
</tr>
<tr>
<td>$y^j_i, \bar{y}^j_i$</td>
<td>helper leaves in $x_i$ gadget for $x^j_i$ and $\bar{x}^j_i$ (resp.) (leaves)</td>
</tr>
<tr>
<td>$\bar{z}^j_i$</td>
<td>$j$th element in sequence $b_j, b'<em>j, \bar{x}^1_i, \bar{y}^1_i, ..., \bar{x}^m_i, \bar{y}^m_i, b</em>{j+1}, b'_{j+1}$ (leaves)</td>
</tr>
<tr>
<td>$F$</td>
<td>SAT formula</td>
</tr>
</tbody>
</table>

### Inconsistent Rooted Triple Set Closure

**INSTANCE:** An inconsistent rooted triple set $R$.

**SOLUTION:** $R$’s closure $\overline{R} = \{ t : R \vdash t \}$.

By the observation above, computing the closure is equivalent to solving the following decision problem for each of the $O(|L|^3)$ triples $t \in (\ell t^2) \times L$.

### Inconsistent Rooted Triple Set Entailment

**INSTANCE:** An inconsistent rooted triple set $R$ and a rooted triple $t$.

**QUESTION:** Does $R \vdash t$, i.e., does there exists a consistent triple set $R' \subset R$ satisfying $R' \vdash t$?

Although there is no finite set of inference rules that are complete [6], there are only three possible inference rules inferring from two triples [6].

**Definition 5.** The three dyadic inference rules ($\forall \ p, q, o, p', o' \in L$) are:

$$
\begin{align*}
\{pq|o, \ p\alpha|o\} \vdash pp'|o \\
\{pq|o, \ q\alpha|o'\} \vdash \{pq|o, \ po|o'\} \\
\{pp'|o, \ oo'|p\} \vdash \{pp'|o, \ oo'|p'\}
\end{align*}
$$
A type of graph (distinct from hypergraphs discussed below) that will be used in the hardness proof is the *Ahograph* \[1\], which is defined for a given triple set \( R \) and leaf set \( L \).

**Definition 6.** For a given triple set \( R \) and leaf set \( L \), the *Ahograph* \([R, L]\) is the following undirected edge-labeled graph:
- its vertex set equals \( L \);
- for every triple \( pqo \in R \), if \( p, q, o \in L \), then there exists an \( \{p, q\} \) with label \( o \).

For a hypergraph \((V, A)\), the corresponding *Ahograph* is the Ahograph \([\text{triples}(A), V]\).

To avoid confusion with the nodes of the hypergraph, we refer to the Ahograph’s nodes and edges as *A-nodes* and *A-edges*.

### 2.2 Directed Hypergraphs

Definitions of paths and cycles in hypergraphs are subtler and more complicated than the corresponding definitions for graphs (see \[10\]). We adopt versions of Gallo et al. \[7\]’s definitions, simplified for the special case in which every hyperarc has exactly one tail and two heads.

**Definition 7.** A 1-2-directed hypergraph (or simply hypergraph) \( H = (V, A) \) consists of a set of nodes \( V \) and a set of 1-2-hyperarcs \( A \). A 1-2-hyperarc (or 1-2-directed hyperedge\(^2\), or simply hyperarc or arc) is an ordered pair \( a = (u, \{v, v'\}) \in V \times \binom{V}{2} \), with \( u, v, v' \) all distinct, which we denote by \( u \to \{v, v'\} \). Let \( t(a) = u \) be \( a \)'s tail and \( h(a) = \{v, v'\} \) be \( a \)'s heads. A node with out-degree 0 is a *sink*.

**Definition 8.**
- A *simple path* from \( u_0 \) to \( u_\ell \) is a sequence of distinct 1-2-hyperarcs \( P = (a_1, \ldots , a_\ell) \), where \( u_0 = t(a_1) \), \( u_\ell \in h(a_\ell) \), and \( t(a_{k+1}) \in h(a_k) \) for all \( k \in [\ell-1] \). The length \( |P| = \ell \) is the number of arcs.
- A *cycle* is a simple path having \( h(a_\ell) \ni t(a_1) \). An arc \( a_k \in P \) having one of its heads be the tail of some earlier arc \( a_{k'} \) of \( P \), i.e., where \( \exists a_{k'} \in P : k' < k \) and \( h(a_k) \ni t(a_{k'}) \), is a back-arc. A simple path is cycle-free or acyclic if it has no back-arcs, and is cyclic otherwise. More generally, a set \( A' \subseteq A \) is cyclic if it is a superset of some cycle, and acyclic otherwise.

**Definition 9.** In general directed hypergraphs (i.e., with no restrictions on arcs’ numbers of heads and tails), a node \( v \) is *B-connected\(^3\)* to \( u_0 \) if \( v = u_0 \) or (generating such B-connected nodes bottom-up, through repeated application of this definition) if there is a hyperarc \( a \) with \( v \in h(a) \) and every node \( t(a) \) is B-connected to \( u_0 \). A path \( P \) from \( u_0 \) to \( u_\ell \) is a *B-hyperpath* if \( u_\ell \) is B-connected to \( u_0 \) (using only the arcs \( a \in P \)).

Due to the following observation, for the remainder of this paper any use of the term “path” will be understood to mean “B-hyperpath”.

**Observation 10.** If all arcs are 1-2-hyperarcs, then every simple path is also a B-hyperpath.

Via the hypergraph representation used in our hardness proof for INCONSISTENT ROOTED TRIPLE SET ENTAILMENT below, we also obtain hardness results for the following problem formulations as a by-product.

\(^1\) We choose to define the Ahograph as a multigraph whose edges each have exactly one label, rather than the more common definition as a graph whose edges each have a set of labels.

\(^2\) Called a 2-directed F-hyperarc in \[14\], extending definitions introduced by Gallo et al. \[7\].

\(^3\) Note also that Gallo et al. \[7\] defines *B-hyperarc* simply to mean an arc \( a \) having \( |h(a)| = 1 \).
Acyclic B-Hyperpath Existence in a 1-2-Hypergraph

**INSTANCE:** A 1-2-directed hypergraph $H = (V, A)$ and nodes $u, v \in V$.

**QUESTION:** Does there exist an acyclic B-hyperpath in $H$ from $u$ to $v$?

We want to define an optimization version of the problem where the objective is to minimize path $P$’s length $|P|$, but since a given problem solution may contain no solutions at all (it may be *infeasible*, specifically if $v$ is not B-connected to $u$), we obtain the following somewhat awkward definition. Note that defining the cost of an infeasible solution to be infinity is consistent with the convention that $\min \emptyset = \infty$.

**Min Acyclic B-Hyperpath in a 1-2-Hypergraph**

**INSTANCE:** A 1-2-directed hypergraph $H = (V, A)$ and nodes $u, v \in V$.

**SOLUTION:** A B-hyperpath $P$ in $H$.

**MEASURE:** $P$’s length $|P|$, (i.e., its number of hyperarcs), if $P$ is a feasible solution (i.e., an acyclic B-hyperpath from $u$ to $v$), and otherwise infinity.

Alternatively, we can formulate a “promise problem” [8] special case of the minimization problem, restricted to instances admitting feasible solutions.

**Min Acyclic B-Hyperpath in a B-Connected 1-2-Hypergraph**

**INSTANCE:** A 1-2-directed hypergraph $H = (V, A)$ and nodes $u, v \in V$, where the $v$ is B-connected to $u$.

**SOLUTION:** An acyclic B-hyperpath $P$ in $H$ from $u$ to $v$.

**MEASURE:** $P$’s length $|P|$.

### 3 The Construction

#### 3.1 High-level Strategy

We will prove that **Inconsistent Rooted Triple Set Entailment** is NP-Complete by reduction from 3SAT, using a construction similar to that of [3] (see also [4]) for the problem of deciding whether a specified pair of nodes in a directed graph are connected by an induced path.\(^4\) So, given a SAT formula $F$, we must construct a problem instance $(R, t)$ such that $R \vdash t$ iff $F$ is satisfiable. Intuitively, we want to define $R$ in such a way that it will be representable as a graph (or rather, as a directed hypergraph), whose behavior will mimic that of the induced subgraph problem.

In slightly more detail, the instance $(R, t)$ that we define based on $F$ will have a structure that makes it representable as a certain directed hypergraph. This hypergraph (see Fig. 1) will play an intermediate role between $(R, t)$ and $F$, yielding a two-step reduction between the three problems. In particular, we will show:

1. A path $P$ (from $\alpha \beta$ to $c_{m+1}\gamma$) determines a truth assignment $v(\cdot)$, and vice versa.
2. $P$ will be acyclic iff $v(\cdot)$ satisfies $F$.
3. An acyclic path $P$ (or an acyclic superset of it) determines a consistent subset $R' \subset R$ entailing $t = \alpha \beta | \gamma$, and vice versa.
4. Hence $R'$ will be consistent and entail $\alpha \beta | \gamma$ iff $P$ is acyclic iff $v(\cdot)$ satisfies $F$.

---

\(^4\) That problem becomes trivial if either the graph is undirected or the *induced* constraint is removed.
A core idea of our construction and proof is a correspondence between rooted triples and the hyperarcs (all 1-2-hyperarcs), which renders them mutually definable in terms of one another. Each of a hyperarc encodes a rooted triple, rather than a constraint of the more general form \( p,q \rightarrow \{o, p, q\} \). Thus we can write \( A \) = \( \{pq \rightarrow \{po, qo\} : pq | o \in R\} \). Indeed, we can simply identify them with one another as follows.

**Definition 11.** For a triple \( pq | o \), the corresponding hyperarc is \( arc(pq | o) = pq \rightarrow \{po, qo\} \); conversely, for a 1-2-hyperarc \( pq \rightarrow \{po, qo\} \), the corresponding triple is \( triple(pq \rightarrow \{po, qo\}) = pq | o \). For a triple set \( R' \), we write \( arcs(R') \) to denote the same set \( R' \), with but its members treated as arcs, and similarly in reverse, for an arc set \( A' \), we write triples(\( A' \)).
Deciding the Closure of Inconsistent Rooted Triples Is NP-Complete

Given this, we can also give a more abstract correspondence.

Definition 12. For a 1-2-hyperarc \( u \to \{v, v'\} \), the corresponding triple is triple \((u \to \{v, v'\}) = v \oplus v' [v \cap v']\), where \( \oplus \) denotes symmetric difference. We also combine the two models' syntax, writing \( u[o] \) to denote \( pq[o] \) when \( u = pq \), i.e., when hyperarc \( u \to \{v, v'\} = \text{arc}(pq[o]). \)

This leads to the following equivalent restatements of the second dyadic inference rule (recall Def. 5) in forms that will sometimes be more convenient.

Observation 13. The first inference of dyadic inference rule (1) can be stated as:

\[
\begin{align*}
\{pq \to \{po, qo\}, \quad \{qo, o', oo'\} \} & \quad \{pq \to \{po', qo'\}\} & (\forall p, q, o, o' \in L) \\
\{uk-1[o], \quad uk[o']\} & \quad \{uk-1[o] \} & (\forall uk-1, uk \in V, o \in uk \text{ s.t. } |uk-1 \cap uk| = 1)
\end{align*}
\]

We emphasize again the following two related facts about the meaning of an arc \( pq \to \{po, qo\} \in A \):

1. If \( T \) is a tree with \( p, q, o \in L(T) \) and \( pq[o] \in r(T) \), then lowest common ancestors \( po \) and \( qo \) are equal, i.e., they refer to the same node in \( T \).

2. Yet \( po \) and \( qo \) are two distinct A-nodes (in \( V \)) of the hypergraph \( H \).

That is, "turning on" triple \( pq[o] \) (by adding it to the triple set \( R' \)) has the effect of causing the hypergraph nodes \( po \) and \( qo \) to thence refer to the same tree node (in any tree displaying \( R' \)).

3.3 Defining \( L \) and \( R \)

Let the SAT formula \( F \) on variables \( x_1, \ldots, x_n \) consist of \( m \) clauses \( C_j \), each of the form \( C_j = (\bar{x}_i^j \lor \bar{x}_j^i \lor \bar{x}_k^j) \) or \( C_j = (\bar{x}_i^j \lor \bar{x}_j^i \lor \bar{x}_k^j) \), where each literal \( \bar{x}_i^j \) has the form either \( x_i \) or \( \bar{x}_i \) for some \( i \).

We define the leaf set \( L \) underlying \( R \) as \( L = L_1 \cup L_2 \cup L_3 \cup L_4 \), where:

\[
\begin{align*}
L_1 &= \bigcup_{j \in [m], i \in [n]} \{ x_i^j, \bar{x}_i^j, y_i^j, \bar{y}_i^j \} \quad (4mn \text{ leaves}) \quad \text{5} \\
L_2 &= \bigcup_{j \in [m+1]} \{ b_i, \bar{b}_i \} \quad (2n + 2 \text{ leaves}) \\
L_3 &= \bigcup_{j \in [m]} \{ c_j, d_j \} \quad (2m \text{ leaves}) \\
L_4 &= \{ \alpha, \beta, \gamma \} \quad (3 \text{ leaves})
\end{align*}
\]

For each variable \( x_i \) in \( F \), we create a gadget consisting of two parallel length-2\( m+2 \) paths intersecting at their first and last nodes but otherwise node-disjoint (see Fig. 2a), where the path taken will determine the variable’s truth value. The rooted triples in \( R \) corresponding to variable \( x_i \)'s gadget are:

- On its positive side:
  \( \{ b_i, c_{i1}^j, \bar{b}_i, c_{i1}^j, x_i^j, \bar{x}_i^j, y_i^j, \bar{y}_i^j, \ldots, x_i^{m-1}y_i^m | b_{i+1}, y_i^m | b_{i+1}, \bar{y}_i^m | b_{i+1} \} \)

- On its negative side:
  \( \{ b_i, c_{i2}^j, \bar{b}_i, c_{i2}^j, x_i^j, \bar{x}_i^j, y_i^j, \bar{y}_i^j, \ldots, \bar{x}_i^{m-1}\bar{y}_i^m | \bar{x}_i^m, \bar{y}_i^m | b_{i+1}, \bar{x}_i^m | b_{i+1}, \bar{y}_i^m | b_{i+1} \} \)

For each clause \( C_j = (\bar{x}_i^j \lor \bar{x}_j^i \lor \bar{x}_k^j) \) in \( F \), we create a gadget consisting of three (or two, in the case of a two-literal clause) parallel length-3 paths, intersecting in their first and fourth nodes, followed by one additional (shared) edge (see Fig. 2b), where the path taken (the witness path) will correspond to which of \( C_j \)'s literal satisfies the clause (or one among them, in the case of multiple true literals). The second node of \( C_j \)'s witness path (of the

\footnote{Alternatively, we could create such nodes only corresponding to actual appearances of variables in clauses, i.e., \( L_1 = \bigcup_{j \in [m], i \in C_j} \{ x_i^j, y_i^j \} \cup \bigcup_{j \in [m], i \in C_j} \{ \bar{x}_i^j, \bar{y}_i^j \} \quad (\leq 3m \text{ leaves}) \).}
(a) Variable gadget for $x_i$. Any path passing through this gadget (drawn left to right) has two options, taking its negative (lower) side, making $x_i$ true, or its positive (higher) side, making $x_i$ false. That is, the truth value corresponding to the path is the one making the literals in the nodes on the unused side true. Intuitively, a path traversing one of the gadget’s two sides renders all the literals appearing within that side’s nodes unusable. Note that the rightmost node ($b_i+1b_i+1$) is also (for each $i < n$) the leftmost node of $x_{i+1}$’s gadget.

(b) Clause gadget for $C_j = (x_{j1} \lor \bar{x}_{j2} \lor \bar{x}_{j3})$, which is followed (drawn outside the shaded region) by node $c_{j+1}d_{j+1}$ (or $c_{m+1}\gamma$, in the case of $j = m$). Any path passing through this gadget (drawn right to left) has three options: going up, straight across, or down, each corresponding to one choice among $C_j$’s three possible witness paths. The arrow from the witness path’s witness node, say, $c_j\hat{x}_{j1}$, to a node $\bar{x}_{j2}b_{j2}$ lying within one of the two sides of $x_i$’s gadget (and outside the shaded region) represents the appearance of $x_i$ in $C_j$: the 1-2-hyperarc that arrow is constituent of forces an acyclic path taking this witness path to have taken the opposite side of $x_i$’s gadget.

Figure 2 Gadgets used in the reduction. Each pair of arrows drawn forking from the same tail node represents one 1-2-hyperarc. Sink nodes have dashed borders and are shaded lighter (gray) than non-sink nodes (blue). The clause gadget nodes that point to variable gadget nodes and the variable gadget nodes that can be pointed to by them are both drawn with thick borders.
Deciding the Closure of Inconsistent Rooted Triples Is NP-Complete

form \( c_j \tilde{x}_i \), and corresponding to the appearance of literal \( \tilde{x}_i \) is its witness node. The rooted triples in \( R \) corresponding to clause \( C_j \)’s gadget are:

- \( \{ c_j d_j | x_j, c_j x_j | y_j, c_j y_j | c_{j+1} \} \), for each positive appearance of a variable \( x_i \) in \( C_j \)
- \( \{ c_j d_j | \tilde{x}_j, c_j \tilde{x}_j | \tilde{y}_j, c_j \tilde{y}_j | c_{j+1} \} \), for each negative appearance of a variable \( x_i \) in \( C_j \)
- \( c_j c_{j+1} | d_{j+1} \), if \( j < m \)

Finally, \( R \) has the following triples connecting the pieces together, connecting the source node \( \alpha \beta \) to a chained-together series of variable gadgets, the last of which is connected (via an intermediate node) to the first of a chained-together series of clause gadgets, the last of which is connected to the destination node \( c_{m+1} \gamma \):

- \( \{ \alpha \beta | b_1, \beta b_1 | b'_1 \} \)
- \( \{ b_{n+1} b'_{n+1} | c_1, b_{n+1} c_1 | d_1 \} \)
- \( c_m c_{m+1} | \gamma \)

It is important to remember that all these connections are 1-2-hyperarcs. Sometimes both heads will be nodes within variable and clause gadgets, but in most cases one of the two heads will be a sink node whose only role is to permit the hyperarc to conform to the required structure.

4 The Proof

Clearly \textsc{Inconsistent Rooted Triple Set Entailment} is in \( \text{NP} \); if we guess the subset \( R' \subset R \), then we can verify both that \( R' \) is consistent and that \( R' \vdash t \) by executing Aho et al. [1]'s polynomial-time BUILD algorithm on \( R' \) [6]. \textsc{Min Acyclic B-Hyperpath in a 1-2-Hypergraph} is as well: guess the path, and check that it is acyclic.

Now we prove hardness, arguing that \( R \) contains a consistent subset entailing \( \alpha \beta | \gamma \) iff \( H \) contains an acyclic path \( P \) from \( \alpha \beta \) to \( c_{m+1} \gamma \) iff \( F \) admits a satisfying assignment \( v(\cdot) \), in two steps. Due to lack of space, the proofs are deferred to the full version.

4.1 Acyclic Path \( \Leftrightarrow \) Satisfying Truth Assignment

First we argue that acyclic paths correspond to satisfying truth assignments.

\textbf{Lemma 14.} There is an an acyclic path \( P \) from \( \alpha \beta \) to \( c_{m+1} \gamma \) iff \( F \) admits a satisfying truth assignment \( v(\cdot) \).

Thus we have proven:

\textbf{Theorem 15.} \textsc{Acyclic B-Hyperpath Existence in a 1-2-Hypergraph} is \( \text{NP-Complete} \).

Since an infeasible solution is defined to have infinite cost, an algorithm with \textit{any} approximation factor would allow us to distinguish between positive and negative problem instances, which immediately implies:

\textbf{Corollary 16.} Approximating \textsc{Min Acyclic B-Hyperpath in a 1-2-Hypergraph} to within any factor is \( \text{NP-hard} \).

Even if we restrict ourselves to problem instances admitting feasible solutions, this “promise problem” [8] special case is hard to approximate within any reasonable factor.

\textbf{Corollary 17.} \textsc{Min Acyclic B-Hyperpath in a B-Connected 1-2-Hypergraph} is \( \text{NP-hard} \) to approximate to within factor \( |V|^{1-\epsilon} \) for all \( \epsilon > 0 \).
Second, to extend the reduction to inconsistent rooted triple set entailment, we argue that $H$ is a faithful representation of $R$ in the sense that acyclic paths from $\alpha\beta$ to $c_{m+1}\gamma$ (or acyclic supersets of such paths) correspond to consistent subsets entailing $\alpha\beta|\gamma$, and vice versa.

### 4.2 Consistent Entailing Subset $\iff$ Acyclic Path

We prove this direction via two lemmas, proving that the set of triples corresponding to an acyclic path are consistent and entail $\alpha\beta|\gamma$, respectively.

- **Lemma 18.** If there is an acyclic path $P \subseteq A$ from $\alpha\beta$ to $c_{m+1}\gamma$, then $R' = \text{triples}(P)$ is consistent.

- **Lemma 19.** If there is an acyclic path $P \subseteq A$ from $\alpha\beta$ to $c_{m+1}\gamma$, then $R' = \text{triples}(P)$ entails $\alpha\beta|\gamma$.

Thus we have:

- **Corollary 20** ($\Leftarrow$). If there is an acyclic path $P \subseteq A$ from $\alpha\beta$ to $c_{m+1}\gamma$, then $R' = \text{triples}(P)$ is consistent and entails $\alpha\beta|\gamma$.

### 4.3 Consistent Entailing Subset $\Rightarrow$ Acyclic Path

Now we argue for the reverse direction, proving through a series of lemmas that if there is no acyclic $\alpha\beta-c_{m+1}\gamma$ path, then there will be no consistent triple subset entailed $\alpha\beta|\gamma$.

- **Lemma 21.** Let $A' \subseteq A$. Suppose there exists a cyclic path $P \subseteq A'$ from $\alpha\beta$ to $c_{m+1}\gamma$. Then $R' = \text{triples}(A')$ is inconsistent.

Most of the remainder of this subsection will be dedicated to showing constructively that if $A'$ contains no path from $\alpha\beta$ to $c_{m+1}\gamma$ at all, cyclic or otherwise, then $R'$ does not entail $\alpha\beta|\gamma$. We do so by showing that in the case of such a (consistent) $R'$, there exist trees displaying $R' \cup \{\alpha\beta|\gamma\}$. Therefore assume w.l.o.g. that $R'$ is consistent and maximal in the sense that adding any other triple of $R$ to it would either make $R'$ inconsistent or would introduce an $\alpha\beta-c_{m+1}\gamma$ path in $A' = \text{arcs}(R')$.

Observe that the missing arcs $A^x = A - A'$ can be thought of as the (source side to sink side) cross arcs of a cut separating source $\alpha\beta$ and sink $c_{m+1}\gamma$. In the following argument we will refer to hypergraph $H^\gamma = (V \cup \{\gamma\alpha\}, A' \cup \text{arc}(\gamma\alpha|\beta))$ and its corresponding Ahograph $G^\gamma$.

Recalling the construction of $H$, there are three types of places where the absent cross-arcs $A^x$ could be located: within a clause gadget, within a variable gadget, or elsewhere, i.e., forced arcs (viz., connecting arcs $a_1, ..., a_4$ or arcs with tail of the form $c_jc_{j+1}$ following a clause $C_j$’s gadget). There is one special subcase, which we give a name to.

- **Definition 22.** We call $A^x$ degenerate if $A^x$ lies within a variable $x_i$’s gadget, $|A^x| = 2$, and exactly one of its members has the form $\text{arc}(b_ib_i|x_i^2)$. (Its other member must by definition lie within the $x_i$ gadget’s opposite side.)

We deal with all cases besides an degenerate $A^x$ in the following lemma.

- **Lemma 23.** Let $R'$ be consistent. Suppose there is no path $P \subseteq A'$ from $\alpha\beta$ to $c_{m+1}\gamma$, and that $A^x$ is non-degenerate. Then $R'$ does not entail $\alpha\beta|\gamma$. 

ISAAC 2018
The problematic situation is when exactly one of the two arcs is outgoing from $b_ib'_i$. In this case, their absence deletes only one of the Ahograph’s two A-edges between the pair $\{b_i, b'_i\}$, which does not disconnect the graph, meaning BUILD will fail.

We have been arguing that if a consistent $R'$ entails $\alpha \beta | \gamma$ then $\text{arcs}(R')$ must contain an acyclic path from $\alpha \beta$ to $\epsilon_{m+1}\gamma$. Now we refine this to a slightly weaker (yet strong enough) implication: if a consistent $R'$ entails $\alpha \beta | \gamma$, then a slightly different consistent $R^+$ will too, and an acyclic path must exist within $\text{arcs}(R^+)$. This implies:

- **Corollary 24** ($\Rightarrow$). If there is a consistent $R'$ entailing $\alpha \beta | \gamma$ then there exists an acyclic path $P$.

Combining the Corollary 20 and 24 with Theorem 15, we conclude:

- **Theorem 25. Inconsistent Rooted Triple Set Entailment is NP-Complete.**

And because computing the closure reduces to deciding whether $R \vdash t$ for $O(|L|^3)$ triples $t$, we also have:

- **Corollary 26. Inconsistent Rooted Triple Set Closure is NP-hard.**

References


Computing Vertex-Disjoint Paths in Large Graphs Using MAOs

Johanna E. Preißer
Institute of Mathematics, TU Ilmenau, Germany

Jens M. Schmidt
Institute of Mathematics, TU Ilmenau, Germany

Abstract
We consider the problem of computing \( k \in \mathbb{N} \) internally vertex-disjoint paths between special vertex pairs of simple connected graphs. For general vertex pairs, the best deterministic time bound is, since 42 years, \( O(\min\{k, \sqrt{n}\}m) \) for each pair by using traditional flow-based methods.

The restriction of our vertex pairs comes from the machinery of maximal adjacency orderings (MAOs). Henzinger showed for every MAO and every \( 1 \leq k \leq \delta \) (where \( \delta \) is the minimum degree of the graph) the existence of \( k \) internally vertex-disjoint paths between every pair of the last \( \delta - k + 2 \) vertices of this MAO. Later, Nagamochi generalized this result by using the machinery of mixed connectivity. Both results are however inherently non-constructive.

We present the first algorithm that computes these \( k \) internally vertex-disjoint paths in linear time \( O(m) \), which improves the previously best time \( O(\min\{k, \sqrt{n}\}m) \). Due to the linear running time, this algorithm is suitable for large graphs. The algorithm is simple, works directly on the MAO structure, and completes a long history of purely existential proofs with a constructive method. We extend our algorithm to compute several other path systems and discuss its impact for certifying algorithms.

2012 ACM Subject Classification Mathematics of computing → Graph theory, Mathematics of computing → Graph algorithms

Keywords and phrases Computing Disjoint Paths, Large Graphs, Vertex-Connectivity, Linear-Time, Maximal Adjacency Ordering, Maximum Cardinality Search, Big Data, Certifying Algorithm


Funding This research is supported by the grant SCHM 3186/1-1 (270450205) from the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation).

Acknowledgements We wish to thank On-Hei S. Lo for pointing out a connection between MAOs and Mader’s proof about pendant pairs.

1 Introduction

Vertex-connectivity is a fundamental parameter of graphs that, by a result due to Menger [12], can be characterized by the existence of internally vertex-disjoint paths between vertex pairs. Thus, much work has been devoted to the following question: Given a number \( k \), a simple graph \( G = (V, E) \), and two vertices of \( G \), compute \( k \) internally vertex-disjoint paths between these vertices if such paths exist. Despite all further efforts, the traditional flow-based approach by Even and Tarjan [3] and Karzanov [7] gives still the best deterministic bound \( O(\min\{k, \sqrt{n}\}m) \) for this task, where \( n := |V| \) and \( m := |E| \).
Our research is driven by the question whether \( k \) internally vertex-disjoint paths can be computed faster deterministically. This question has particular impact for large graphs, as we aim for linear-time algorithms. We have no general answer, but show for specific pairs of vertices that this can actually be done using maximal adjacency orderings (MAOs, also known under the name maximum cardinality search). MAOs order the vertices of a graph and can be computed in time \( O(n + m) \) \cite{18} (we will define MAOs in detail in Section 2).

One of the key properties of MAOs is that their last vertices are highly vertex-connected, i.e., have pairwise many internally vertex-disjoint paths. In more detail, let \( G \) be a simple unweighted graph of minimum degree \( \delta \) and let \( < \) be a MAO of \( G \). Then \( < \) decomposes \( G \) into edge-disjoint forests \( F_1, \ldots, F_m \) in a natural way (we will give the precise background on MAOs and such forest decompositions later). Let a subset of vertices be \( k \)-connected if \( G \) contains \( k \) internally vertex-disjoint paths between every two vertices of this subset. Henzinger proved for every \( 1 \leq k \leq \delta \) that the last \( \delta - k + 2 \) vertices of \( < \) are \( k \)-connected \cite{6}.

In order to appreciate Henzinger’s result, it is important to mention that its special case \( k = \delta \) alone was predated by many results in the (weaker) realm of edge-connectivity: a well-known line of research \cite{14, 4, 17} proved that the last two vertices of \( < \) are \( \delta \)-edge-connected. In fact, we exhibit the following forgotten link to a result by Mader \cite{10, 9} in 1971, who used a preliminary variant of MAOs over one decade before MAOs were introduced and proved that their last two vertices are even \( \delta \)-connected. In 2006, Nagamochi generalized all the mentioned results as follows.

\textbf{Theorem 1} \cite{13, 15, Thm. 2.28}. Let \( < \) be a MAO of a simple graph \( G \) and let \( F_1, \ldots, F_m \) be the forests into which \( < \) partitions \( E \). For every two vertices \( s \) and \( t \) that are in the same component of some \( F_k, G \) contains \( k \) internally vertex-disjoint paths between \( s \) and \( t \).

Theorem 1 specializes to Henzinger’s result by taking the component \( T_k \) of \( F_k \) that contains the last vertex of \( < \) (this tree contains the last \( \delta - k + 2 \) vertices of \(<\)). Its proof depends heavily on the machinery of mixed connectivity, and so does its most general statement (which we omit here, although all our results extend to this setting). Theorem 1 may be seen as the currently strongest result on MAOs regarding vertex-connectivity. However, all proofs known so far about vertex-connectivity in MAOs (including the ones by Henzinger and Nagamochi) are non-constructive and thus do not give any faster algorithm than the flow-based one for the initial question of computing internally vertex-disjoint paths.

The main result of this paper is an algorithm that computes the \( k \) paths of Theorem 1 in linear time \( O(n + m) \). This improves upon the previously best time \( O(\min\{k, \sqrt{n}\}m) \). To our surprise, its key idea is simple; the details of its correctness proof however are subtle. We therefore explain the algorithm in two incremental variants: The slightly weaker variant in Section 3 computes internally vertex-disjoint paths between one vertex \( s \) and a fixed set of \( k \) vertices of the forest decomposition; it does so by performing a right-to-left sweep through the MAO, in which the \( k \) paths are switched cyclically whenever one of the \( k \) paths would be lost. Section 4 then invokes two of these computations (one for \( s \) and one for \( t \)) in parallel in order to obtain our main result. We show also how the computation can be extended to find the \( k \) internally vertex-disjoint paths between a vertex and a vertex set, and between two vertex sets, whose existence was shown by Menger \cite{12}.

It is not easy to quantify for how many vertex pairs our faster algorithm can be applied. If we require \( \delta \) internally vertex-disjoint paths, there are \( \delta \)-regular graphs for which the only component of \( F_3 \) consists of one vertex pair joined by an edge and \( F_{\delta+1} = \cdots = F_m = \emptyset \). In this case, we can apply our algorithm only to a single vertex pair. However, in practice, many more of these sets occur and each of them may have a much larger size. If \( \delta < \delta \) internally vertex-disjoint paths are sufficient, all pairs of a much larger set of size \( \delta - k + 2 \) can be taken (even in the worst case), at the expense of the linearly decreased pairwise connectivity \( k \).
Certifying Algorithms

Being able to compute $k$ internally vertex-disjoint paths has a benefit that purely existential proofs and algorithms that only argue about vertex separators do not have: It certifies the connectivity between the two vertices. For related problems on edge-connectivity, this has already been used to make algorithms certifying (in the sense of [11]).

The perhaps most prominent such result is the minimum cut algorithm of Nagamochi and Ibaraki [14], which refines the work of Mader [10, 9], and was simplified by Frank [4] and by Stoer and Wagner [17]. This algorithm computes iteratively a MAO and then contracts the last two δ(-edge)-connected vertices of it. For unweighted multigraphs, this is easily made certifying by storing the $k$ edge-disjoint paths between these last two vertices in every step; the global $k$-edge-connectivity then follows by transitivity. In fact, the desired $k$ edge-disjoint paths for every MAO can be obtained by just taking, for every $1 \leq i \leq k$, the unique $s$-$t$-path in the tree $T_i$ of $F_i$ that contains $t$. Using more involved methods, Arikati and Mehlhorn [1] made the algorithm of Nagamochi and Ibaraki certifying even for weighted graphs, again without increasing the quadratic asymptotic running time and space.

For the problem of recognizing $k$-connectivity, linear-time certifying algorithms are known for every $k \leq 3$ [19, 16]. For arbitrary $k$, the best known deterministic certifying algorithm is still the traditional flow-based one [3, 5], which achieves a running time of $O((k + \sqrt{n})k\sqrt{\text{tim}})$. By using a geometric characterization of graphs, also a non-deterministic certifying algorithm with running time $O(n^{5/2} + k^{5/2}n)$ is known [8]. For designing faster certifying algorithms, finding a good certificate for $k$-connectivity seems to be the crucial open graph-theoretic problem, even when $k$ is fixed:

**Open Problem.** For every $k \in \mathbb{N}$, find a small and easy-to-verify certificate that proves the $k$-vertex-connectivity of simple graphs.

Our main result plays the same important role for certifying the vertex-connectivity between two vertices, as $s$-$t$-flows do for certifying the edge-connectivity between $s$ and $t$ in the results described above. For example, the 2-approximation algorithm for vertex-connectivity [6] by Henzinger can be made certifying using our new algorithm.

2 Maximal Adjacency Orderings

Throughout this paper, our input graph $G = (V, E)$ is simple, unweighted and of minimum degree $\delta$. We assume standard graph theoretic notation as in [2]. A maximal adjacency ordering $\prec$ of $G$ is a total order $1, \ldots, n$ on $V$ such that, for every two vertices $v < w$, $v$ has at least as many neighbors in $\{1, \ldots, v - 1\}$ as $w$ has. For ease of notation, we always identify the vertices of $G$ with their position in $\prec$.

Every MAO $\prec$ decomposes $G$ into edge-disjoint forests $F_1, \ldots, F_m$ (some of which may be empty)\(^1\) as follows: If $v > 1$ is a vertex of $G$ and $w_1 < \cdots < w_l$ are the neighbors of $v$ in $\{1, \ldots, v - 1\}$, the edge $\{w_i, v\}$ belongs to $F_i$ for all $i \in \{1, \ldots, l\}$. For every $i$, the graph $(V, F_i)$ is an edge-maximal forest of $G \setminus \{E(F_1), \ldots, E(F_{i-1})\}$ (we refer to [15, Section 2.2] for a proof). For the sake of conciseness, we identify this forest with its edge set $F_i$. The partition of $E$ into the non-empty forests is called the forest decomposition of $\prec$. For vertices $v < w$, we say $v$ is left of $w$. If there is an edge between $v$ and $w$, we call this a left-edge of $w$.

For any $k$, we allow to compute $k$ internally vertex-disjoint paths between any two vertices that are contained in a tree $T_k$ of the forest $F_k$. Hence, throughout the paper, let $s > 1$ be an arbitrary but fixed vertex of $G$ and let $k$ be a positive integer that is at most the number

\(^1\) In fact, every forest $F_i$ that satisfies $i > n$ is empty, as $G$ is simple.
of left-edges of $s$. The vertex $s$ will be the start vertex of the $k$ internally vertex-disjoint paths to find (the end vertex will be left of $s$). E.g., if we choose $s$ as the last vertex of the MAO (or any other vertex with at least that many left-edges), $k$ can be chosen as any value that is at most the degree of vertex $n$; in particular, $k$ can be chosen arbitrary in the range $1, \ldots, \delta$, as claimed in the introduction.

For $i \in \{1, \ldots, k\}$, let $T_i$ be the component of $F_i$ that contains $s$. As $i \leq k$, $T_i$ is a tree on at least two vertices. Let the smallest vertex $r_i$ of $T_i$ with respect to $< \in$ be the root of $T_i$. For the purpose of this paper, it suffices to consider the subgraph of $G$ induced by the edges of $T_1, \ldots, T_k$.

Lemma 2 ([15, Lemma 2.25]). Let $i \in \{1, \ldots, k\}$. Then $V(T_i)$ consists of the consecutive vertices $r_i, r_i + 1, \ldots, w$ in $<$ such that $s \leq w$. Moreover, for each vertex $v \in T_i \setminus \{r_i\}$, the vertex set $\{v, r_i + 1, \ldots, v\}$ induces a connected subgraph of $T_i$.

Hence, for every $i \in \{1, \ldots, k\}$, every vertex $v > r_i$ of $T_i$ has exactly one left-edge that is in $T_i$ and thus at least $i$ left-edges that are in $G$. Let $\text{left}_i(v)$ be the end vertex of the left-edge of $v$ in $F_i$. The root $r_i$ of $T_i$ has left-degree exactly $i - 1$, as if it had more, $r_i$ would have a left-edge in $F_i$ and thus not be the root of $T_i$ and, if it had less, the left-degree of $r_i + 1$ cannot be at least $i$, as this violates the MAO (this uses that $G$ is simple). We conclude that $r_1 < r_2 < \cdots < r_k$. Thus, the definition of $F_i$ and Lemma 2 imply the following corollary.

Corollary 3. Let $i < j \leq k$ and let $v$ be a vertex with $r_j < v < s$. Then $v$ is in $T_j$ and $T_i$, $r_i \leq \text{left}_i(v) < \text{left}_j(v) < v$ and $r_j \leq \text{left}_j(v)$.

For a vertex-subset $S \subseteq V$, let $\overline{S} := V \setminus S$. For convenience, we will denote sets $\{v\}$ by $\overline{v}$. For a vertex-subset $S \subseteq V$, a set of paths is $S$-disjoint if no two of them intersect in a vertex that is contained in $S$. Thus, $V$-disjointness is the usual vertex-disjointness and a set of paths is $\overline{v}$-disjoint if every two of them intersect in either the vertex $v$ or not at all. We represent paths as lists of vertices. The length of a path is the number of edges it contains. For a path $A$, let $\text{end}(A)$ be the last vertex of this list and, if the path has length at least one, let $\text{sec}(A)$ be the second to last vertex of this list.

3 The Loose Ends Algorithm

We first consider the slightly weaker problem of computing $k$ internally vertex-disjoint paths between $s$ and the root set $\{r_1, \ldots, r_k\}$. We will extend this to compute $k$ internally vertex-disjoint paths between two vertices in the next section.

Lemma 4. Algorithm 1 computes $k \overline{s}$-disjoint paths in $T_1 \cup \cdots \cup T_k$ from $s$ to $\{r_1, \ldots, r_k\}$ in time $O(|E(T_1 \cup \cdots \cup T_k)|) \subseteq O(n + m)$.

The outline of our algorithm is as follows. We initialize each $A_i$ to be the path that consists of the two vertices $s$ and $\text{left}_i(s)$ (in that order). The vertices $\text{left}_i(s)$ are marked as active; throughout the algorithm, let a vertex be active if it is an end vertex of an unfinished path $A_i$.

So far the $A_i$ are $\overline{s}$-disjoint. We aim for augmenting each $A_i$ to $r_i$. Step by step, for every active vertex $v$ from $s$ down to $r_1$, we will modify the $A_i$ to longer paths, similar as in sweep line algorithms from computational geometry. The modification done at an active vertex $v$ is called a processing step. From a high-level perspective, the end vertices of several paths $A_i$ may be replaced or augmented by new end vertices $w$ such that $r_i \leq w < v$ during the processing step of $v$. Such vertices $w$ are again marked as active, which results in a
Algorithm 1 LooseEnds\((G, \prec, s, k)\).

1: for all \(i\) do \(\triangleright\) initialize all \(A_i\)
2: \(A_i := (s, \text{left}_i(s))\)
3: Mark \(\text{left}_i(s)\) as active
4: while there is a largest active vertex \(v\) do \(\triangleright\) process \(v\)
5: Let \(j_1 < j_2 < \cdots < j_i\) be the indices of the paths \(A_{j_i}\) that end at \(v\)
6: for \(i := 2\) to \(l\) do \(\triangleright\) replace end vertices
7: Replace \(\text{end}(A_{j_i})\) with \(\text{left}_{j_i-1}(\text{sec}(A_{j_i}))\)
8: Mark \(\text{left}_{j_i-1}(\text{sec}(A_{j_i}))\) as active
9: Perform a cyclic downshift on \(A_{j_1}, \ldots, A_{j_l}\) \(\triangleright\) \(A_{j_i} := A_{j_{i+1}}, A_{j_{i+1}} := A_{j_i}\)
10: if \(v = r_{j_i}\) then \(\triangleright r_{j_i}\) is reached
11: \(A_{j_i}\) is finished
12: else
13: Append \(\text{left}_{j_i}(v)\) to \(A_{j_i}\) \(\triangleright\) append predetermined vertex
14: Mark \(\text{left}_{j_i}(v)\) as active
15: Unmark \(v\) from being active
16: Output \(A_1, \ldots, A_k\)

continuous modification of each \(A_i\) to a longer path. By the above restriction on \(v\), each path \(A_i\) will have strictly decreasing vertices in \(\prec\) throughout the algorithm. At the end of the processing step of \(v\), we unmark \(v\) from being active.

Let \(v\) be the active vertex that is largest in \(\prec\). Assume that \(v\) is the end vertex of exactly one \(A_i\). If \(v = r_i\), \(A_i\) is finished. Otherwise, we append the vertex \(\text{left}_i(v)\) to \(A_i\) (see Algorithm 1). The important aspect of this approach is that the index of the path \(A_i\) predetermines the vertex that augments \(A_i\). Clearly, this way \(A_i\) will reach \(r_i\) at some point, according to Lemma 2.

However, if at least two paths end at \(v\), this approach does not ensure vertex-disjointness. Let \(A_{j_1}, \ldots, A_{j_l}\) be these \(l \geq 2\) paths and assume \(j_1 < j_2 < \cdots < j_l\). We first replace the end vertex \(v\) of \(A_{j_i}\) with the vertex \(\text{left}_{j_i-1}(\text{sec}(A_{j_i}))\) for all \(i \neq 1\). We will show that these modified end vertices are strictly smaller than \(v\), which will re-establish the vertex-disjointness. The key idea of the algorithm is then to switch the indices of the \(l\) paths appropriately such that the appended vertices are again predetermined by the path index.

Let a cyclic downshift on \(A_{j_1}, \ldots, A_{j_l}\) replace the index of each path by the next smaller index of a path in this set (where the next smaller index of \(j_i\) is \(j_{i+1}\)), i.e. we set \(A_{j_i} := A_{j_{i+1}}\) for every \(i \neq l\) and then replace \(A_{j_i}\) with the old path \(A_{j_i}\). We perform a cyclic downshift on \(A_{j_1}, \ldots, A_{j_l}\). Note that we did not alter the path \(A_{j_l}\) (which was named \(A_{j_1}\) before) yet. If \(v = r_{j_i}\), \(A_{j_i}\) is finished; otherwise, we append the vertex \(\text{left}_{j_i}(v)\) to \(A_{j_i}\). See Algorithm 1 for a description of the algorithm in pseudo-code. Figure 1 shows a run of Algorithm 1.

We prove the correctness of Algorithm 1. Before the processing step of any active vertex \(v\), the \(A_i\) satisfy several invariants, the most crucial of which are that they are \(\{v + 1, \ldots, s - 1\}\)-disjoint and that the vertices of every \(A_i\) are decreasing in \(\prec\). In detail, we have the following invariants.

\section*{Invariants.} Let \(v < s\) be the largest active vertex, or \(v := 0\) if there is no active vertex left.

Before processing \(v\), the following invariants are satisfied for every \(1 \leq i \leq k\):

1. The vertices of \(A_i\) start with \(s\) and are strictly decreasing in \(\prec\).
2. The path \(A_i\) is finished if and only if \(\text{end}(A_i) > v\). In this case, \(\text{end}(A_i) = r_i\).

   If \(A_i\) is not finished, \(r_i \leq \text{end}(A_i) < v\) and the last edge of \(A_i\) is in \(T_i\).
(a) A MAO of a graph $G$ and its forests $F_1$ (green), $F_2$ (red, dashed) and $F_3$ (blue, dotted).

(b) Paths $A_1$ (green), $A_2$ (red, dashed) and $A_3$ (blue, dotted) after the initialization phase and processing vertex 11. The paths $A_2$ and $A_3$ end at the largest active vertex 10.

(c) After processing vertex 10, the paths $A_2$ and $A_3$ have been shifted, which is here depicted by a color change. The last vertex of $A_2$ is then replaced, while $A_3$ is extended in $F_3$.

(d) After processing 9, the largest active vertex is 6.

(e) After shifting and extending $A_1$ and $A_3$, all three paths meet at the largest active vertex 4.

(f) Downshift: The old path $A_3$ is now $A_2$, the old $A_2$ is now $A_1$ and the old $A_1$ is now $A_3$.

(g) After processing root $r_3 = 3$, $A_2$ and $A_3$ are shifted and $A_3$ is finished.

(h) After processing the roots $r_2 = 2$ and $r_1 = 1$, the paths $A_1$ and $A_2$ are finished.

**Figure 1** A run of Algorithm 1 on the graph depicted in (a) when $s = 12$ and $k = 3$. 
(3) \( \sec(A_i) > v \)
(4) Every vertex \( w \in A_i \) satisfying \( v < w < s \) is not contained in any \( A_j \neq A_i \).
(5) \( A_i \subseteq T_1 \cup \cdots \cup T_k \)

We first clarify the consequences. Invariant (2) implies that the algorithm has finished all paths \( A_i \), precisely after processing \( r_1 \), and that every \( A_i \) ends at \( r_1 \). The Invariants (1) and (3) are necessary to prove Invariant (4), which in turn implies that the \( A_i \) are \( \{v+1, \ldots, s-1\} \)-disjoint before processing an active vertex \( v \). Hence, the final paths \( A_i \) are \( \bar{\pi} \)-disjoint. With Invariant (5) this gives the claim of Lemma 4.

It remains to prove Invariants (1)–(5). Immediately after initializing \( A_1, \ldots, A_k \), the next active vertex is \( \text{end}(A_k) < s \). It is easy to see that all five invariants are satisfied for \( v = \text{end}(A_k) \), i.e. before processing the first active vertex. We will prove that processing any largest active vertex \( v \) preserves all five invariants for the active vertex \( v' \) that follows \( v \) (where \( v' := 0 \) if \( v \) is the only remaining active vertex). For this purpose, let \( A'_l \) be the path with index \( i \) immediately before processing \( v' \) and let \( A_j \) be the path with index \( i \) before processing \( v' \); by hypothesis, the paths \( A_i \) satisfy all invariants for \( v \).

For Lines 7 and 13 in the processing step of \( v \), we have to prove the existence of \( \left<j_{i-1} (\sec(A_j)) \right> \) and \( \left< j_{i} (v) \right> \) respectively. In Line 7, we have \( i \geq 2 \) and \( \text{end}(A_{j_{i-1}}) = v \) as can be seen in the pseudo-code. Then Invariant (2) implies that \( A_{j_{i-1}} \) is not finished and \( v = \text{end}(A_{j_{i-1}}) = \left<j_{i-1} (\sec(A_j)) \right> \). Thus, \( \left<j_{i-1} (\sec(A_j)) \right> \) exists. In Line 13, we have \( v \neq r_{j_i} \text{ and } \text{end}(A_{j_i}) = v \) (here, \( A_{j_i} \) refers by definition to the path with index \( j_i \) before the cyclic downshift; note this is not the path dealt with in Line 13). Then Invariant (2) implies that \( r_{j_i} \leq v \). This proves \( r_{j_i} < v \) and the existence of \( \left<j_{i} (v) \right> \).

We prove \( v' < v \) next. Consider the vertices that are newly marked as active in the processing step of \( v \). According to Line 5 of Algorithm 1, every such vertex is the new end vertex of some path \( A_{j_i} \) with end vertex \( v \) that was modified in the processing step of \( v \) (we do not count index transformations as modifications). There are exactly two cases how \( A_{j_i} \) may have been modified, namely either by Line 7 (then \( 2 \leq i \leq l \) and \( \left<j_{i-1} (\sec(A_j)) \right> \) is the vertex that is newly marked as active) or by Line 13 (then \( \left< j_{i} (v) \right> \) is the vertex that is newly marked as active); in particular, \( A_{j_{i-1}} \) was not modified by both lines. In the first case, \( A_{j_{i}} \) satisfies Invariant (2) before the processing step of \( v \) by hypothesis. In fact, we have \( r_{j_i} \leq v \), as \( v < r_{j_i} \), implies that \( A_{j_{i}} \) is finished and since \( \text{end}(A_{j_{i}}) > v \) would contradict \( \text{end}(A_{j_{i}}) = v \).

Hence, the last edge of \( A_{j_{i}} \) is in \( T_{j_i} \), which shows \( v = \left<j_{i-1} (\sec(A_j)) \right> \). Since \( j_{i-1} < j_i \) by Line 5 and due to Corollary 3, we conclude \( \left<j_{i-1} (\sec(A_j)) \right> < v \). In the second case, Corollary 3 implies \( \left< j_{i} (v) \right> < v \). Thus, in both cases, every new active vertex is strictly smaller than \( v \), which proves \( v' < v \).

This gives Invariant (1), as every \( A'_{j_i} \) starts with \( s \) and every new vertex is left of its predecessor in the path by Corollary 3.

For Invariant (2), consider the path \( A'_{i} \) for any \( i \). First, assume that \( A'_{i} \) is finished. Then either \( A_{j_{i}} \) is finished or \( v = r_{i} \), according to Line 11 of Algorithm 1 in the processing step of \( v \). In the former case, \( A_j \) satisfies Invariant (2) for \( v \) and so does \( A'_{i} \) for \( v' < v \). In the latter case, we have \( v' < v = r_{i} \text{ and } \text{end}(A'_{i}) = \text{end}(A_{j_{i}}) = v \).

Second, assume that \( A'_{i} \) was not modified in the processing step of \( v \) and is not finished. Then \( \text{end}(A'_{i}) < v \), as every path with end vertex at least \( v \) is modified or finished in the processing step of \( v \) or finished before. In particular, processing \( v \) did not change the index of \( A_{j_{i}} = A'_{i} \). As \( A_j \) satisfies Invariant (2) for \( v \) by hypothesis, the only condition of Invariant (2) that may be violated for \( v' \) is \( \text{end}(A'_{i}) \leq v' \). However, as \( \text{end}(A'_{i}) < v \) was marked as active in some previous step of Algorithm 1 and since \( v' \) is the largest active vertex, \( \text{end}(A'_{i}) \leq v' \). Thus, \( A'_{i} \) satisfies Invariant (2) for \( v' \).

Third, assume that \( A'_{j_i} \) was modified in the processing step of \( v \) and is not finished. Then \( A'_{j_i} \) was modified either by Line 7 or 13. If \( A'_{j_i} \) was modified by Line 7, we have \( i < l \) and
2 \leq l after the cyclic downshift, as the path \( A_j \) is not modified by Line 7. In addition, we know \( \text{end}(A_j) = \text{left}_j(\text{sec}(A_j)) \leq \text{left}_{j+1}(\text{sec}(A_{j+1})) = v \) by Corollary 3 and that the last edge of \( A_j \) is in \( T_{ji} \). Thus, \( r_{ji} \leq \text{end}(A_j) \). If \( A_j \) was modified by Line 13, we have \( i = l \) and \( r_{ji} \leq \text{left}_j(v) = \text{end}(A_j) \) by Corollary 3. Then the last edge of \( A_j \) is in \( T_{ji} \). In both cases, \( \text{end}(A_j) \) is active before processing \( v' \) and it follows \( \text{end}(A_j) \leq v' \).

For Invariant (3), assume to the contrary that \( \text{sec}(A_j) \leq v' \). Since \( v' < v < \text{sec}(A_j) \) for all \( j \in \{1, \ldots, k\} \), a new end vertex was appended to \( A_j \) in the processing step of \( v \) (the end vertex was not replaced, as this would not have changed \( \text{sec}(A_j) \)). This must have been done in Line 13 of Algorithm 1 and we conclude \( v' < v = \text{sec}(A_j) \), which contradicts the assumption.

For Invariant (4), consider Line 7 of the processing step of \( v \). As showed in the proof of \( v' < v \) above, we have \( \text{left}_{j-1}(\text{sec}(A_j)) < v \) for all \( 1 < i \leq l \). Thus, Invariants (1) and (3) imply that exactly the path \( A_j \) of the paths \( A_1', \ldots, A_k' \) contains \( v \).

Invariant (5) follows directly from the definition of \( \text{left}_i \). This concludes the correctness part of the proof of Lemma 4.

So far we have shown an algorithmic proof for the existence of \( k \) internally vertex-disjoint paths from \( s \) to the roots \( r_1, \ldots, r_k \). It remains to show the running time for Lemma 4. At every point in time, we maintain the order \( A_1 < \cdots < A_l \) on our \( i \leq k \) internally vertex-disjoint paths, where \( i \) is the index of the root vertex \( r_i \) that will be visited next. This ordered list can be updated in constant time after each cyclic downshift by modifying the position of one element.

Let \( v \) be the currently active vertex and let \( r_i \leq v \) be the root vertex that will be visited next. Consider the ordered list of unfinished paths \( A_1 < \cdots < A_i \) just before invoking Line 5. For Line 5, we need to sort the subset \( A_i, \ldots, A_j \) \( (j \leq i) \) of such paths paths ending at \( v \) according to \( < \). In order to do this, we run through the \( i \) paths \( A_1 < \cdots < A_i \) in that order, check for each entry whether its end vertex is \( v \), and if so, append it to the sorted list \( A_{j_1} < A_{j_2} < \ldots \). Since \( v \) has precisely \( i \) (or \( i - 1 \) in case of \( v = r_i \)) left-edges in \( T_1 \cup \cdots \cup T_k \subseteq G \), this running time is upper-bounded by the number of such left-edges plus one. Summing the number of these left-edges for every visited \( v \) thus gives a running time bound of \( O(|E(T_1 \cup \cdots \cup T_k)|) \) for all invocations of Line 5. Since the algorithm visits every edge only a constant number of times, this implies a total running time of \( O(|E(T_1 \cup \cdots \cup T_k)|) = O(n+m) \).

## 4 Computing Vertex-Disjoint Paths Between Two Vertices

We use the algorithm of the last section to prove our following main result.

> **Theorem 5.** Let \( t < s \) be a vertex in \( T_k \). Then \( k \) internally vertex-disjoint paths between \( s \) and \( t \) can be computed in time \( O(|E(T_1 \cup \cdots \cup T_k)|) \subseteq O(n+m) \).

This theorem is directly implied by the following lemma.

> **Lemma 6.** Let \( t < s \) be a vertex in \( T_k \). Then there are \( k \) paths \( A_1, \ldots, A_k \) with start vertex \( s \) and \( k \) paths \( B_1, \ldots, B_k \) with start vertex \( t \) such that \( \text{end}(A_i) = \text{end}(B_i) \) for every \( i \) and \( \{A_1 \cup B_1, \ldots, A_k \cup B_k\} \) is a set of \( k \) internally vertex-disjoint paths from \( s \) to \( t \). Moreover, all paths are contained in \( T_1 \cup \cdots \cup T_k \) and can be computed by Algorithm 2 in time \( O(|E(T_1 \cup \cdots \cup T_k)|) \).

A first idea would be to use the loose ends-algorithm twice, once for the start vertex \( s \) and once for the start vertex \( t \), in order to find the paths \( A_i \) and \( B_i \) for all \( i \). However, in general
this is bound to fail. In some cases, the union of both outputs is a graph in which \( s \) and \( t \) are not \( k \)-connected. A second attempt may try to finish two paths \( A_i \) and \( B_j \) whenever they end at the same active vertex. However, this may fail when \( i \neq j \), as then two single paths \( A_{i'} \) and \( B_{j'} \) may remain that end at the respective roots \( r_{i'} \) and \( r_{j'} \) such that \( B_{j'} \) cannot be extended to \( r_v \) without violating the index scheme of Invariant (2).

We will nevertheless use Algorithm 1 to prove Lemma 6, but in a more subtle way, as outlined next. First, we compute the paths \( A_1, \ldots, A_k \) with start vertex \( s \) using Algorithm 1, until the largest active vertex \( v \) is less or equal \( t \) (i.e. the parts of the \( A_i \) between \( s \) and \( t \) are just computed by Algorithm 1). As soon as \( v \leq t \), we additionally construct a second set of paths \( B_1, \ldots, B_k \) with start vertex \( t \) using Algorithm 1.

The main difference to Algorithm 1 from this point on is that we extend the paths \( A_i \) and the paths \( B_i \) in parallel (i.e. we take the largest active vertex of both running constructions) such that, after the processing step of \( v \), the vertex \( v \) is not contained in any two paths \( A_i \) and \( B_j \) with \( i \neq j \). This ensures the vertex-disjointness.

If no \( A \)-path or no \( B \)-path ends at \( v \), we again just perform Algorithm 1; then at most one path contains \( v \) after the processing step. Otherwise, some \( A \)-path and some \( B \)-path ends at \( v \). After the processing step at \( v \), we want to have exactly two paths \( A_j \) and \( B_i \) (i.e. having the same index) that end at \( v \); such a pair of paths is then finished. In order to ensure this, we choose \( j \) as the largest index such that \( A_j \) or \( B_j \) ends at \( v \) before processing \( v \). If both \( A_j \) and \( B_j \) end at \( v \), we perform one processing step of Algorithm 1 at \( v \) for the \( A \)-paths and the \( B \)-paths, respectively, which implies that no other path is ending at \( v \).

Otherwise, exactly one of the paths \( A_j \) and \( B_j \) ends at \( v \), say \( A_j \). Then \( B_j \) is not finished, as we finish only paths having the same index, and the last edge of \( B_j \) is in \( F_j \). By assumption, there is an index \( i < j \) such that \( B_i \) ends at \( v \). We then apply a processing step of Algorithm 1 (including a cyclic downshift) on \( B_j \) and all \( B \)-paths that end at \( v \), and one on all \( A \)-paths, respectively. Then the new paths \( A_j \) and \( B_j \) (due to cyclic downshifts, these correspond to the former \( A \)- and \( B \)-paths with lowest index ending at \( v \)) end at \( v \) afterward, but no other \( A \)- or \( B \)-path, as desired. Note that the replacement of the last edge of \( (\text{the old}) \) \( B_j \), which did not end at \( v \) but, say, at a vertex \( w \), may cause \( w \) to be active although neither an \( A \)-path nor a \( B \)-path ends at \( w \).

For a precise description of the approach, see Algorithm 2. The following observations follow directly from Algorithm 2.

**Observation 1.** Throughout Algorithm 2 the paths \( A_1, \ldots, A_k, B_1, \ldots, B_k \) satisfy the following properties.

1. For every \( i \in \{1, \ldots, k\} \), \( A_i \) and \( B_i \) are both finished or both unfinished.
2. As long as the largest active vertex is larger than \( t \), \( B_1 = B_2 = \cdots = B_k = \{t\} \).
3. The end vertex of every unfinished path is active.

Before the processing step of any active vertex \( v \), the paths \( A_i \) and \( B_i \) satisfy several invariants, the most crucial of which are that they are \( \{v + 1, \ldots, s - 1\} \backslash \{t\} \)-disjoint and that the vertices of every \( A_i \) and \( B_i \) are decreasing in \( < \).

**Invariants.** Let \( v < s \) be the largest active vertex, or \( v := 0 \) if there is no active vertex left. Before processing \( v \), the following invariants are satisfied for every \( 1 \leq i \leq k \):

1. \( A_i \) starts with \( s \), \( B_i \) starts with \( t \), and the vertices of both paths are strictly decreasing in \( < \).
2. The paths \( A_i \) and \( B_i \) are finished if and only if \( v < \text{end}(A_i) = \text{end}(B_i) \). If \( A_i \) and \( B_i \) are not finished, then \( r_i \leq \text{end}(A_i) \leq v, r_i \leq \text{end}(B_i) \leq v \), and the last edge of \( A_i \) as well as the last edge of \( B_i \) (if \( B_i \) has length at least 1) are in \( T_i \).
Algorithm 2 MatchingEnds($G, <, s, t, k$).\footnote{$t$ is a vertex in $T_k$, $t < s$}

1. \textbf{for all $i$ do} \hspace{1em} \triangleright \text{initialize all } A_i \text{ and } B_i
2. \quad A_i := (s, \text{left}_i(s))
3. \quad \text{Mark } \text{left}_i(s) \text{ as active}
4. \quad B_i := (t)
5. \quad \text{Mark } t \text{ as active}
6. \textbf{while} there is a largest active vertex $v$ \textbf{do} \hspace{1em} \triangleright \text{process } v
7. \quad \textbf{if } v = t \textbf{ then} \hspace{1em} \triangleright \text{initialize all } A_i
8. \quad \quad \textbf{for all } i \textbf{ do}
9. \quad \quad \quad \text{if } \text{end}(A_i) = t \textbf{ then}
10. \quad \quad \quad \quad A_i, B_i \text{ are finished}
11. \quad \quad \quad \textbf{else}
12. \quad \quad \quad \quad \text{Append } \text{left}_i(t) \text{ to } B_i
13. \quad \quad \quad \quad \text{Mark } \text{left}_i(t) \text{ as active}
14. \quad \quad \quad \textbf{end}
15. \quad \quad \text{Unmark } t \text{ from being active}
16. \quad \textbf{else}
17. \quad \quad \textbf{for all pairs } (i_1, i_2) \text{ of consecutive indices } i_1 < i_2 \text{ in } I_A \cup \{j\} \textbf{ do}
18. \quad \quad \quad \text{Replace } \text{end}(A_{i_2}) \text{ with } \text{left}_i(\text{sec}(A_{i_2})) \hspace{1em} \triangleright \text{replace ends}
19. \quad \quad \quad \text{Mark } \text{left}_i(\text{sec}(A_{i_2})) \text{ as active}
20. \quad \quad \textbf{end}
21. \quad \quad \textbf{for all pairs } (i_1, i_2) \text{ of consecutive indices } i_1 < i_2 \text{ in } I_B \cup \{j\} \textbf{ do}
22. \quad \quad \quad \text{Replace } \text{end}(B_{i_2}) \text{ with } \text{left}_i(\text{sec}(B_{i_2})) \hspace{1em} \triangleright \text{replace ends}
23. \quad \quad \quad \text{Mark } \text{left}_i(\text{sec}(B_{i_2})) \text{ as active}
24. \quad \quad \textbf{end}
25. \quad \textbf{Perform a cyclic downshift on all } A_i \text{ with } i \in I_A \cup j
26. \quad \textbf{Perform a cyclic downshift on all } B_i \text{ with } i \in I_B \cup j
27. \quad \textbf{if } v = \text{end}(A_j) = \text{end}(B_j) \textbf{ then} \hspace{1em} \triangleright \text{if and only if } I_A \neq \emptyset \neq I_B
28. \quad \quad \text{if } A_j, B_j \text{ are finished}
29. \quad \quad \quad \text{Append } \text{left}_i(v) \text{ to } A_j \hspace{1em} \triangleright \text{append predetermined vertex}
30. \quad \quad \quad \text{Mark } \text{left}_i(v) \text{ as active}
31. \quad \quad \textbf{else if } v = \text{end}(A_j) \textbf{ then}
32. \quad \quad \quad \text{Append } \text{left}_i(v) \text{ to } A_j \hspace{1em} \triangleright \text{append predetermined vertex}
33. \quad \quad \text{Mark } \text{left}_i(v) \text{ as active}
34. \quad \quad \textbf{else if } v = \text{end}(B_j) \textbf{ then}
35. \quad \quad \quad \text{Append } \text{left}_i(v) \text{ to } B_j \hspace{1em} \triangleright \text{append predetermined vertex}
36. \quad \quad \quad \text{Mark } \text{left}_i(v) \text{ as active}
37. \quad \quad \textbf{end}
38. \quad \text{Unmark } v \text{ from being active}
39. \quad \textbf{end}
40. \textbf{end}
41. \textbf{end}
42. \textbf{end}
43. \textbf{end}
44. Output $A_1, \ldots, A_k, B_1, \ldots, B_k$
Assume Invariant (2). Then no finished path is modified while processing $v$. Let

Assume Invariants (1) and (3). Then, for every $A_i \neq A_j$ and $B_i \neq B_j$. If $w \in A_i \cap B_i$, $A_i$ and $B_i$ are finished with

$w = \text{end}(A_i) = \text{end}(B_i)$. Invariant (5) settles the first part of the second claim of Lemma 6. We continue with further consequences of some of these invariants, which can be used to prove the invariants for the next largest active vertex $v'$ after processing $v$.

**Observation 2.** Let $v < s$ be the largest active vertex, or $v := 0$ if there is no active vertex left. Before processing $v$, we have the following observations:

1. Assume Invariants (1) and (3). Then, for every $1 \leq i \leq k$, all vertices of the paths $A_i$ and $B_i$ except $\text{end}(A_i)$ and $\text{end}(B_i)$ are greater than $v$ before processing $v$.
2. Assume Invariant (2). Then no finished path is modified while processing $v$, as Algorithm 2 modifies $A_i$ or $B_i$, $1 \leq i \leq k$, only if at least one of them ends at $v$.
3. Assume Invariants (2) and (3). Then the largest active vertex after processing $v > 0$ is smaller than $v$.

Due to space constraints, we omit the proofs of the Invariants (1)-(5) and Observation 2. As in the loose ends algorithm, the running time of Algorithm 2 is upper bounded by $O(|E(T_1 \cup \cdots \cup T_k)|)$ and thus by $O(n + m)$, as it suffices to visit every edge in the trees $T_1, \ldots, T_k$ a constant number of times.

### 4.1 Variants

Several variants of Menger's theorem [12] are known. Instead of computing $k$ paths between two vertices, we can compute paths between a vertex and a set of vertices (fan variant) and between two sets of vertices (set variant). Our algorithm extends to these variants.

**Theorem 7.** Let $G$ be a simple graph and $s$ and $T_1, \ldots, T_k$ be defined as in Section 2.

(i) (Fan variant) Let $T = \{t_1, \ldots, t_k\}$ be a subset of $V$ such that $r_i \leq t_i < s$ for every $i$. Then $k$ internally vertex-disjoint paths between $s$ and $T$ can be computed in time $O(|E(T_1 \cup \cdots \cup T_k)|) \subseteq O(n + m)$.

(ii) (Set variant) Let $T = \{t_1, \ldots, t_k\}$ and $S = \{s_1, \ldots, s_k\}$ be disjoint vertex sets such that $r_i \leq t_i < s$ and $s_i \leq s$ for every $i$. Then $k$ internally vertex-disjoint paths between $S$ and $T$ can be computed in time $O(|E(T_1 \cup \cdots \cup T_k)|) \subseteq O(n + m)$.

Let $\alpha : V \rightarrow \mathbb{N}^+$ be a weight function. In the area of mixed connectivity, a set of paths connecting two vertices $s$ and $t$ of $G$ is called $\alpha$-independent if every vertex $v \notin \{s, t\}$ is contained in at most $\alpha(v)$ of these paths. For suitable multigraphs $G$, Nagamochi [13] generalized Theorem 1 by showing that these contain $k$ $\alpha$-independent $s$-$t$-paths. Algorithm 2 can be modified to compute also these paths without increasing its running time, by replacing the two cyclic downshifts by a more complicated algorithm that transforms the path indices.
References

An $O(n^2 \log^2 n)$ Time Algorithm for Minmax Regret Minsum Sink on Path Networks

Binay Bhattacharya  
School of Computing Science, Simon Fraser University, Burnaby, Canada

Yuya Higashikawa  
School of Business Administration, University of Hyogo, Kobe, Japan

Tsunehiko Kameda  
School of Computing Science, Simon Fraser University, Burnaby, Canada

Naoki Katoh  
School of Science and Technology, Kwansei Gakuin University, Sanda, Japan

Abstract

We model evacuation in emergency situations by dynamic flow in a network. We want to minimize the aggregate evacuation time to an evacuation center (called a sink) on a path network with uniform edge capacities. The evacuees are initially located at the vertices, but their precise numbers are unknown, and are given by upper and lower bounds. Under this assumption, we compute a sink location that minimizes the maximum “regret.” We present the first sub-cubic time algorithm in $n$ to solve this problem, where $n$ is the number of vertices. Although we cast our problem as evacuation, our result is accurate if the “evacuees” are fluid-like continuous material, but is a good approximation for discrete evacuees.

2012 ACM Subject Classification  
Networks → Network algorithms, Mathematics of computing → Graph algorithms, Applied computing → Transportation

Keywords and phrases  
Facility location, minsum sink, evacuation problem, minmax regret, dynamic flow path network

Digital Object Identifier  
10.4230/LIPIcs.ISAAC.2018.14

Related Version  
A full version of the paper is available at [3], https://arxiv.org/abs/1806.00814.

Acknowledgements  
This work is supported in part by NSERC of Canada Discovery Grant, in part by JST CREST (JPMJCR1402), and in part by JSPS KAKENHI Grant-in-Aid for Young Scientists (B) (17K12641).

1 Introduction

The goal of evacuation planning is to evacuate all the evacuees to some sinks, optimizing a certain objective function [8, 16]. Some aspects of such planning can be modeled by dynamic flow in a network [6] whose vertices represent the places where the evacuees are initially located and the edges represent possible evacuation routes. Associated with each edge is the transit time across the edge and its capacity in terms of the number of people who can enter it per unit time. Evacuation starts from all vertices at the same time.

A completion time k-sink, a.k.a. minmax k-sink, is a set of $k$ sinks that minimizes the time until every evacuee has moved to a sink. If the edge capacities are uniform, it is easy to compute a completion time 1-sink in path networks in linear time [5, 10]. Mamada et al. [16]
solved this problem for the tree networks with non-uniform edge capacities in $O(n \log^2 n)$ time, when the sink is constrained to be at a vertex. Higashikawa et al. proposed an $O(n \log n)$ algorithm without this constraint when the edges have the same capacity [12].

The concept of regret was introduced by Kouvelis and Yu [15], to model the situations where optimization is required when the exact values (such as the number of evacuees at the vertices) are unknown, but are given by upper and lower bounds. A particular instance of the set of such numbers, one for each vertex, is called a scenario. The objective is to find a solution which is as good as any other solution in the worst case, where the actual scenario is the most unfavorable. Cheng et al. [5] proposed an $O(n \log^2 n)$ time algorithm for finding a minmax regret 1-sink in path networks with uniform edge capacities. This initial result was soon improved to $O(n \log n)$ [10, 17], and further to $O(n)$ [4]. Bhattacharya and Kameda [4] propose an $O(n \log^4 n)$ time algorithm to find a minmax regret 2-sink on path networks. For the $k$-sink version of the problem, Arumugam et al. [1] give two algorithms, which run in $O(kn^3 \log n)$ and $O(kn^2 (\log n)^k)$ time, respectively. As for the tree networks with uniform edge capacities, Higashikawa et al. [12] propose an $O(n^2 \log^2 n)$ time algorithm for finding a minmax regret 1-sink. Golin and Sandeep [7] recently proposed an $O(\max \{k^2, \log^2 n\} k^2 n^5 \log^5 n)$ time algorithm for finding a minmax regret $k$-sink.

The objective function we adopt in this paper is the aggregate evacuation time, i.e., the sum of the evacuation time of every evacuee, a.k.a. minsum [11]. It is equivalent to minimizing the average evacuation time, and is motivated by the desire to minimize the transportation cost of evacuation and the total amount of psychological duress suffered by the evacuees, etc. It is more difficult than the completion time variety because the objective cost function is not unimodal along the given path. The minimization of the evacuation completion time (resp. aggregate evacuation time) reduces to the center (resp. median) problem, when the edge capacities are infinite, but finite capacities can cause congestion [5] which complicates the problems. To the best of our knowledge very little is known about this problem, except [2, 11, 13]. It is recently shown by Benkoczi et al. [2] that an aggregate time $k$-sink in path networks can be found in $O(kn \log^3 n)$ (resp. $O(kn^2 \log^2 n)$) time, if edge capacities are uniform (resp. nonuniform).

The main contribution of this paper is to find an aggregate time 1-sink that minimizes regret in $O(n^2 \log^2 n)$ time, improving the required time from $O(n^3)$ in [11]. A set of $O(n^2)$ dominating scenarios was identified in [11]. We first compute the aggregate time sinks for these scenarios, then the upper envelope of the “regret functions” of all these scenarios. Finally, we compute the lowest point of the upper envelope, which corresponds to the optimal sink $\mu^*$. We make use of a few novel ideas. One is used in Sec. 4 to compute an aggregate time sink under each of the $O(n^2)$ pseudo-bipartite scenarios [11] in amortized $O(\log^2 n)$ time per sink. Another is used in Sec. 5 to compute the upper envelope of $O(n^2)$ regret functions (with $O(n^3)$ linear segments in total) in $O(n^2 \log^2 n)$ time, taking advantage of a special relationship among the regret functions.

In the next section, we define the terms that are used throughout this paper, and review some known facts which are relevant to later discussions. Sec. 3 discusses preprocessing which makes later operations more efficient. In Sec. 4 we show how to compute an aggregate time sink under scenarios that “dominate” others. We then compute in Sec. 5 an optimum sink that minimizes the max regret. The proofs of some lemmas could not be included due to space limitation. The interested reader is referred to the arXived version [3], which provides the proofs of all the lemmas and formal statements of three algorithms.
2 Preliminaries

2.1 Notations/definitions

Let $P(V, E)$ denote a given path network with vertex set $V = \{v_1, v_2, \ldots, v_n\}$. We assume that the vertices are arranged from left to right horizontally in the index order. For $1 \leq i \leq n - 1$, there is an edge $e_i = (v_i, v_{i+1}) \in E$, whose length is denoted by $d(e_i)$. We write $p \in P$ for any point $p$ (on an edge or vertex) of $P$, and for two points $a, b \in P$, we write $a \prec b$ or $b \succ a$ if $a$ lies to the left of $b$. The distance between them is denoted by $d(a, b)$. If $a$ and/or $b$ lies on an edge, the distance is prorated. The capacity (the upper limit on the flow rate) of each edge is $c$ (a constant), and the transit time is $\tau$ per unit distance. For $1 \leq i \leq j \leq n$, $P[v_i, v_j]$ denotes the subpath of $P$ from $v_i$ to $v_j$.

For vertex $v_i$, $w(v_i) \in \mathbb{R}_+$ (the set of the positive reals) denotes its weight, which represents the number of “evacuees” initially located at $v_i$. Under scenario $s$, vertex $v_i$ has a weight $w^s(v_i)$ such that $\overline{w}(v_i) \leq w^s(v_i) \leq \underline{w}(v_i)$, where $\overline{w}(v_i)$ and $\underline{w}(v_i)$ are assumed to be known. We define the Cartesian product $S \triangleq \prod_{i=1}^{n} [\underline{w}(v_i), \overline{w}(v_i)]$, and consider each member of $S$ as a scenario. Most of the above definitions were introduced in [5].

Our objective function under scenario $s$, $\Phi^s(x)$, is the sum of the evacuation times (sometimes called cost) of all the individual evacuees to point $x$. More formally, for $v_i \prec x \preceq v_{i+1}$ (resp. $v_i \prec x \preceq v_{i+1}$), let $\Phi^s_i(x)$ (resp. $\Phi^s_i(x)$) denote the cost at $x$ for the evacuees from the vertices on $P[v_i, v_{i+1}]$ (resp. $P[v_i, v_{i+1}]$). We thus have $\Phi^s(x) \triangleq \Phi^s_i(x) + \Phi^s_j(x)$. Let $\mu^s \triangleq \text{argmin}_x \Phi^s(x)$ be an aggregate time sink under $s$. Then $R^s(x) \triangleq \Phi^s(x) - \Phi^s(\mu^s)$ is called regret at $x$ under $s$ [15]. We say that scenario $s'$ dominates scenario $s$ at point $x$ if $R^s(x) \geq R^{s'}(x)$ holds. The max regret at $x$ is given by $R_{\text{max}}(x) \triangleq \max_{s \in S} R^s(x)$ [15]. Our goal is to find a $1$-sink, $x = \mu^*$, that minimizes $R_{\text{max}}(x)$.

By $\overline{s}_i$, we denote the scenario under which $w(v_j) = \overline{w}(v_j)$ for all $j \leq i$ and $w(v_j) = \underline{w}(v_j)$ for all $j > i$, where $0 \leq i \leq n$. Similarly, by $\underline{s}_i$ we denote the scenario under which $w(v_j) = \overline{w}(v_j)$ for all $j \leq i$ and $w(v_j) = \underline{w}(v_j)$ for all $j > i$. We call $\overline{s}_i$ and $\underline{s}_i$ bipartite scenarios. Finally, we define weight arrays $\overline{W}[v_i] \triangleq \sum_{k=1}^{i} \overline{w}(v_k)$ and $\underline{W}[v_i] \triangleq \sum_{k=1}^{i} \underline{w}(v_k)$, which can be precomputed in $O(n)$ time for all $i$, $1 \leq i \leq n$.

2.2 Clusters

In order to analyze congestion, in this subsection we review the notion of a cluster [11], and introduce some new related concepts, which play important roles in subsequent discussions. Given a point $x \in P$, which is not the sink, the evacuation flow at $x$ toward the sink is a function of time, in general, alternating between no flow and flow at the rate limited by capacity $c$. A maximal group of vertices that provide uninterrupted flow without any gap forms a cluster. Such a cluster observed on edge $e_{k-1} = (v_{k-1}, v_k)$, arriving from right via $v_k$, is called an $R^s$-cluster with respect to (any point on) $e_{k-1}$, including $v_{k-1}$, but excluding $v_k$. The vertex of such a cluster that is closest to $e_{k-1}$ is called its head vertex. An $L^s$-cluster with respect to $e_k$, including $v_{k+1}$, is similarly defined for evacuees arriving from left toward the sink.

If a cluster $C$ contains a vertex $v$, the cluster is said to carry the evacuees from $v$. We now define particular clusters and cluster sequences.

- $C^s_{R,k}(v_i) \triangleq R^s$-cluster with respect to $e_{k-1}$ that contains vertex $v_i$ ($i \geq k$).
- $C^s_{R,k}$: sequence of all $R^s$-clusters with respect to $e_{k-1}$ ($k = 2, \ldots, n$).
- $C^s_{L,k}(v_i) \triangleq L^s$-cluster with respect to $e_k$ that contains vertex $v_i$ ($i \leq k$).
- $C^s_{L,k}$: sequence of all $L^s$-clusters with respect to $e_k$ ($k = 1, \ldots, n - 1$).
When we name the first (resp. second) term in (4) an aggregate time sink, shares the following property of a point \( s \):

\[
d(v_k, v_i) > \lambda^s(C^s_{R,k}(v_k))/c.
\]

Intuitively, this means that when the first evacuee from \( v_i \) arrives at \( v_k \), all evacuees carried by \( C^s_{R,k}(v_k) \) have left \( v_k \) already. For \( v_k-1 \leq x < v_k \), let us analyze the cost of \( C^s_{R,k}(v_i) \) at \( x \), where \( v_i \geq v_k \). For the \( \lambda^s(C^s_{R,k}(v_i)) \) evacuees to move to \( x \), let us divide the time required into two parts. The first part, called the \textit{intra cost} [2], is the weighted waiting time before departure from the head vertex, \( v_j \), of \( C^s_{R,k}(v_i) \), and can be expressed as

\[
\{ \lambda^s(C^s_{R,k}(v_i)) \}^2/2c.
\]

Intuitively, (2) can be interpreted as follows. As far as the travel time to \( v_j \) and the waiting time at \( v_j \) are concerned, we may assume that all the \( \lambda^s(C^s_{R,k}(v_i)) \) evacuees were at \( v_j \) to start with. Since evacuees leave \( v_j \) at the rate of \( c \), the mean wait time for the evacuees carried by \( C^s_{R,k}(v_i) \) is \( \lambda^s(C^s_{R,k}(v_i))/2c \), and thus the total for all of them is

\[
\lambda^s(C^s_{R,k}(v_i))/2c \times \lambda^s(C^s_{R,k}(v_i)) = \{ \lambda^s(C^s_{R,k}(v_i)) \}^2/2c.
\]

Note that the intra cost does not depend on \( x \), as long as \( v_k-1 \leq x < v_k \). This formula is accurate only when it is an integer, but for simplicity, we adopt (2) as our intra cost [5].

The second part, called the \textit{extra cost} [2], is the total transit time from the head vertex \( v_j \) of \( C^s_{R,k}(v_i) \) to \( x \) for all the evacuees carried by \( C^s_{R,k}(v_i) \), and can be expressed as

\[
d(x, v_j)\lambda^s(C^s_{R,k}(v_i))\tau.
\]

For the evacuees carried by \( C^s_{R,k}(v_i) \), moving to the right, we similarly define its intra cost and extra cost, where \( v_1 \leq v_i \prec x \preceq v_{k+1} \). For \( v_k-1 \leq x < v_k \), we now introduce a cost function for cluster sequence \( C^s_{R,k} \):

\[
\Phi^s_{R,k}(x) \triangleq \sum_{C \in C^s_{R,k}} d(x, v_i)\lambda^s(C)\tau + \sum_{C \in C^s_{R,k}} \lambda^s(C)^2/2c.
\]

We name the first (resp. second) term in (4) \( E^s_{R,k} \) (resp. \( I^s_{R,k} \)). Similarly, for \( x \prec x \preceq v_{k+1} \), we define

\[
\Phi^s_{L,k}(x) \triangleq \sum_{C \in C^s_{L,k}} d(v_i, x)\lambda^s(C)\tau + \sum_{C \in C^s_{L,k}} \lambda^s(C)^2/2c \triangleq E^s_{L,k} + I^s_{L,k}.
\]

When \( v_k \) is clear from the context, or when there is no need to refer to it, we may write \( \Phi^s_{R}(x) \) (resp. \( \Phi^s_{L}(x) \)) to mean \( \Phi^s_{R,k}(x) \) (resp. \( \Phi^s_{L,k}(x) \)). The aggregate of the evacuation times to \( x \) of all evacuees is given by

\[
\Phi^s(x) = \begin{cases} 
\Phi^s_{L,k}(x) + \Phi^s_{R,k+1}(x) & \text{for } v_k \prec x \prec v_{k+1} \\
\Phi^s_{L,k}(x) + \Phi^s_{R,k+1}(x) & \text{for } x = v_k.
\end{cases}
\]

A point \( x \) that minimizes \( \Phi^s(x) \) is called an \textit{aggregate time sink}, a.k.a. \textit{minsum sink}, under \( s \). An aggregate time sink shares the following property of a median [14].

\textbf{Lemma 1 ([13])}. Under any scenario, there is an aggregate time sink at a vertex.

\footnote{It is accurate for fluid-like "evacuees" that is always divisible by capacity \( c \).}
2.3 What is known

► Lemma 2 ([11]). For any given scenario $s \in S$,
(a) We can compute $\{\Phi^*_{S}(v_i), \Phi^*_R(v_i) \mid i = 1, \ldots, n\}$ in $O(n)$ time.
(b) We can compute $\mu^*$ and $\Phi^*(\mu^*)$ in $O(n)$ time.

A scenario $s$ under which all vertices on the left (resp. right) of a vertex have the max (resp. min) weights is called an $L$-pseudo-bipartite scenario [11]. The vertex $v_b$, where $1 \leq b \leq n$, that may take an intermediate weight $w(v_b) \leq w(v_b) \leq \overline{w}(v_b)$, is called the boundary vertex, a.k.a. intermediate vertex [11]. Let $b(s)$ denote the index of the boundary vertex under pseudo-bipartite scenario $s$. We consider the bipartite scenarios, under which $w(v_b) = \underline{w}(v_b)$ and $w(v_b) = \overline{w}(v_b)$, also as special pseudo-bipartite scenarios, and in the former (resp. latter) case, either $b(s) = b - 1$ or $b(s) = b$ (resp. $b(s) = b$ or $b(s) = b + 1$).

The vertices that have the maximum (resp. minimum) weights comprise the max-weighted (resp. min-weighted) part. We define an $R$-pseudo-bipartite scenario symmetrically with the max-weighted part and the min-weighted part reversed. As $w(v_b)$ increases from $\underline{w}(v_b)$ to $\overline{w}(v_b)$, clusters may merge.

Weight $w^*(v_b)$ is called a critical weight, if two clusters with respect to any point merge as $w(v_b)$ increases to become a scenario $s$. Let $S_{L}^\ast$ (resp. $S_{R}^\ast$) denote the set of the L- (resp. $R$-)pseudo-bipartite scenarios that correspond to the critical weights. Thus each scenario in $S_{L}^\ast$ (resp. $S_{R}^\ast$) can be specified by $v_b$ and $w(v_b)$. Let $S^\ast = S_{L}^\ast \cup S_{R}^\ast$.

► Lemma 3 ([11]).
(a) Any scenario in $S$ is dominated at every point $x$ by a scenario in $S^\ast$.²
(b) $|S^\ast| = O(n^2)$, and all scenarios in $S^\ast$ can be determined in $O(n^2)$ time.

2.4 Road map

From now on, we proceed as follows.

(1) Investigate important properties of clusters to prepare for later sections. (Sec. 3)
(2) Compute $\{\mu^s \mid s \in S^\ast\}$ in $O(n^2 \log^2 n)$ time. (Sec. 4)
(3) Compute $R_{\max}(x) = \max \{R^s(x) \mid s \in S^\ast\}$ in $O(n^2 \log^2 n)$ time. (Sec. 5.1) $R_{\max}(x)$ is a piecewise linear function, and can be specified by the set of its bending points.
(4) Find point $x = \mu^*$ that minimizes $R_{\max}(x)$ in $O(n^2)$ time. (Sec. 5.2)

3 Clusters under pseudo-bipartite scenarios

3.1 Preprocessing

Without loss of generality, we concentrate on $R$-clusters for $s \in S_{L}^\ast$, since the other combinations, such as $R$-clusters for $s \in S_{R}^\ast$, etc., can be treated analogously. For $k = 2, \ldots, n$, let $C_{R,k}$ consist of $q^*(k)$ clusters

$$C_{R,k}^\ast = \{C_1, C_2, \ldots, C_{q^*(k)}\},$$

and let $u_i$ be the head vertex of $C_i$, where $v_k = u_1 < \ldots < u_{q^*(k)}$. By (1), $d(u_i, u_{i+1}) \tau > \lambda(C_i)/c$ holds for $i = 1, 2, \ldots, q^*(k) - 1$.

² Not necessarily by the same scenario. The scenario depends on a particular $x$. 
Lemma 4.
(a) For any scenario \( s \in S \), the number of distinct clusters in \( \{C_{R,k}^s \mid k = 2, \ldots, n\} \) is \( O(n) \).
(b) For any scenario \( s \in S \), we can construct \( \{C_{R,k}^s \mid k = 2, \ldots, n\} \) in \( O(n) \) time.

Proof. (a) Consider \( C_{R,k}^s \) in the order \( k = n, n-1, \ldots, 2 \). Cluster sequence \( C_{R,v_n}^s \) consists of just one cluster composed of \( v_n \). Let \( C_{R,k+1}^s = \langle C_1', C_2', \ldots, C_{q'(k+1)}' \rangle \) for some \( k \). Cluster \( C_1 \in C_{R,k}^s \) contains vertex \( v_k \) and possibly the vertices of \( C_1', \ldots, C_h' \) for some \( h \), where \( 0 \leq h \leq q'(k+1) \) and \( h = 0 \) means \( C_1 \) contains just \( v_k \) and no other vertex. Note that \( C_1 \) is new when we go from \( k + 1 \) to \( k \), but the other clusters of \( C_{R,k}^s \), i.e., \( C_2, \ldots, C_{q'(k)}' \) are \( C_{h+1}', \ldots, C_{q'(k+1)}' \). This means that each \( k \) introduces just one new cluster, and thus the number of distinct clusters is \( O(n) \).

(b) Let us construct \( C_{R,k+1}^s \) in the order \( k = n, n-1, \ldots, 2 \). Assume that we have computed \( C_{R,k+1}^s \), and want to compute \( C_1 \). If (1) does not hold between the new singleton cluster \( v_k \) and the first cluster, \( C_1' \), of \( C_{R,k+1}^s \), namely if \( d(v_k,v_{k+1})\tau \leq \lambda^s(\{v_k\})/c \), then \( v_k \) and \( C_1' \) merge to form a single cluster. (1) may become violated for this new cluster and \( C_1' \) in case they also merge. As a result of such chain reaction, if \( v_k \) merges with the first \( h \) clusters in \( C_{R,k+1}^s \) this way, we spend \( O(h) \) time in computing \( C_1 \). Those \( h \) clusters will never contribute to the computation time from now on. If we pay attention to the head vertex, \( u_1 \), of \( C_1 \), it gets absorbed into a larger cluster at most once, and each time such an event takes place, constant computation time incurs.

Computing the extra cost \( E_{R,k}^s \) in (4) is fairly easy, because the extra cost of cluster \( C \) is linear in \( \lambda^s(C) \). The intra costs can also be computed efficiently.

Lemma 5 ([11]). Given a scenario \( s \in S \),
(a) We can compute \( \{E_{R,k}^s, I_{R,k}^s \mid k = 1, \ldots, n-1\} \) in \( O(n) \) time.
(b) We can compute \( \{E_{L,k}^s, I_{L,k}^s \mid k = 2, \ldots, n\} \) in \( O(n) \) time.

Let \( s_0 \triangleq s_0 \) and \( s_M \triangleq s_n \). The following corollary follows easily from Lemmas 4 and 5.

Corollary 6.
(a) There are \( O(n) \) distinct clusters among the cluster sequences in \( \{C_{L,k}^{s_0} \cup C_{L,k}^{s_0} \cup C_{L,k}^{s_M} \cup C_{R,k}^{s_M} \mid k = 1, \ldots, n\} \), and we can compute them in \( O(n) \) time.
(b) We can compute \( \{E_{L,k}^{s_0}, I_{L,k}^{s_0}, E_{L,k}^{s_M}, I_{L,k}^{s_M} \mid k = 1, \ldots, n-1\} \) and \( \{E_{L,k}^{s_0}, I_{L,k}^{s_0}, E_{L,k}^{s_M}, I_{L,k}^{s_M} \mid k = 2, \ldots, n\} \) in \( O(n) \) time.
(c) For each cluster sequence in \( \{C_{L,k}^{s_0} \cup C_{L,k}^{s_0} \cup C_{L,k}^{s_M} \cup C_{R,k}^{s_M} \} \), we can compute the prefix sum of the intra costs in \( O(n) \) time. Thus we can compute the prefix sums of the intra costs for all \( k \) in \( O(n^2) \) time, if we do not repeat the common data.

From now on, we assume that we have precomputed all the data mentioned in Corollary 6.

3.2 Constructing set of pseudo-bipartite scenarios \( S^* \)
Let \( s = s_0 \) in (7). Starting with \( b = k \), we increase \( w(v_b) \) until \( C_{R,k}^{s_0}(v_b) \) merges with the next cluster in \( C_{R,k}^{s_0} \), and record the value of \( b \) and the amount of increase \( \delta \) above \( w(v_b) \) that caused this merger. We repeat this with the newly formed cluster, instead of \( C_{R,k}^{s_0}(v_b) \). If \( \overline{w}(v_b) \) is reached we fix \( w(v_b) = \overline{w}(v_b) \), increment \( b \) and repeat, as long as \( v_b \in C_{R,k}^{s_0}(v_b) \) holds. We will end up with a list

\[
\Delta_{R,k} \triangleq \{(b_1, \delta_{b_1,1}), (b_2, \delta_{b_2,2}), \ldots\},
\]
where \( k \leq b_1 \leq b_2 \leq \cdots \), and for any two adjacent items, \((b_i, \delta_{h,i})\) and \((b_{i+1}, \delta_{k,i+1})\), if \( b_i = b_{i+1} \) then \( \delta_{k,i} < \delta_{k,i+1} \). Intuitively, \((b_i, \delta_{h,i})\) \( \in \Delta_{R,k} \) means that when \( w(v_{b_i}) = w(v_{b_i}) + \delta_{k,i} \) the first cluster of \( \mathcal{C}^v_{R,k}(v_k) \) expands by merging with the next cluster, where \( s \) is the scenario reflecting the weight changes made so far. Fig. 1(a) illustrates some clusters in the beginning of \( \mathcal{C}^v_{R,k}(v_k) \), and Fig. 1(b) shows \( \mathcal{C}^m_{R,k}(v_k) \). We start with \( v_b = v_k \) in \( \mathcal{C}^v_{R,k}(v_k) \) in Fig. 1(a), which is a part of \( \mathcal{C}^v_{R,k} \) that we already have. We increase \( w(v_b) \) by \( \delta_{k,1} \) from \( w^0(v_b) = w(v_b) \) until \( \mathcal{C}^m_{R,k}(v_k) \) expands by merging with the next cluster on its right. This \( \delta_{k,1} \) is obtained by solving 3

\[
\begin{align*}
d(u_1, u_2) &= \{ \lambda^0(\mathcal{C}^v_{R,k}(v_k)) + \delta_{k,1} \}/c. 
\end{align*}
\]

Assuming \( w(v_b) + \delta_{k,1} \leq w(v_b) \), for \( w^*(v_b) = w(v_b) + \delta_{k,1}, \mathcal{C}^v_{R,k}(v_k) \) may merge with the next \( h - 1 \) clusters in \( \mathcal{C}^v_{R,k} \), where \( h \geq 2 \), resulting in a combined cluster \( \mathcal{C}^v_{R,k} \) \( \subset C^v_{R,k} \) under \( s \neq s_0 \), and the first item \((k, \delta_{k,1})\) being created in \( \Delta_{R,k} \). If \( w(v_b) + \delta_{k,1} \leq w(v_b) \), on the other hand, we repeat this operation to find the increment \( \delta_{k,2} \), if any, above \( w(v_b) \) that causes \( \mathcal{C}^v_{R,k} \) to absorb the \( h + 1 \) cluster in \( \mathcal{C}^v_{R,k} \), etc. If \( w(v_b) + \delta_{k,1} > w(v_b) \), on the other hand, we set \( w(v_b) = w(v_b) \) and increment \( b \) by one without recording \( \delta_{k,1} \). When this process terminates, we end up with \( \mathcal{C}^m_{R,k}(v_k) \) in Fig. 1(b), because all the vertices involved now have their max weights, and we will have constructed \( \Delta_{R,k} \). Clearly, each item \((b_j, \delta_{b,j})\) \( \in \Delta_{R,k} \) corresponds to a scenario \( s_j \in \mathcal{S}^*_{L} \) in the following way.

\[
\begin{align*}
w^*(v_i) &= \begin{cases} 
\lambda^0(v_i) & \text{for } 1 \leq i < b_j, \\
\lambda^0(v_i) + \delta_{b,j} & \text{for } i = b_j, \\
\lambda^0(v_i) & \text{for } b_j < i \leq n. 
\end{cases}
\end{align*}
\]

Let \( \mathcal{S}^*_L \) denote the set of scenarios corresponding to the increments in \( \Delta_{R,k} \). It is clear that \( \mathcal{S}^*_L = \cup_{k=1}^n \mathcal{S}^*_{L,k} \). Note that under any \( s \in \mathcal{S}^*_L \), \( \mathcal{C}^m_{R,k}(v(b_s)) \) is the first cluster in \( \mathcal{C}^m_{R,k} \).

**Lemma 7.**

(a) We can compute \( \Delta_{R,k} \) in \( O(|\mathcal{C}^m_{R,k}(v_k)|) \) time, where \( |\mathcal{C}^m_{R,k}(v_k)| \) denotes the number of vertices in cluster \( \mathcal{C}^m_{R,k}(v_k) \).

(b) We can construct \( \{ \Delta_{R,k} \mid k = 2, \ldots, n \} \), hence \( \mathcal{S}^*_L \), in \( O(n^2) \) time.

(c) For each scenario \( s \in \mathcal{S}^*_L \), we can identify the last vertex in \( \mathcal{C}^m_{R,k}(v_k) \) in constant extra time while computing \( \Delta_{R,k} \).

---

3 Let \( u_i \) be as defined after (7) for \( s = s_0 \).

4 The above method to compute \( \Delta_{R,k} \) is presented as a formal algorithm in [3].
4 Computing sinks \( \{ \mu^s \mid s \in S^* \} \)

4.1 Computing \( \{ \Phi^s(x) \mid s \in S^* \} \)

Let us now turn our attention to the computation of the extra and intra costs under the scenarios in \( S_{L,k}^* \). Those under the scenarios in \( S_{R,k}^* \) can be computed similarly. While computing \( \Delta_{R,k} \) as in Sec. 3.2, we can update the extra and intra costs at \( v_k \) under the corresponding scenario \( s \in S_{L,k}^* \) as follows.

When the first increment \( \delta_{k,1} \) causes the merger of the first two clusters in \( S_{R,k}^* \), for example, we subtract the extra cost contributions of those two clusters from \( E_{R,k}^s \), and add the new contribution from the merged cluster in order to compute \( E_{R,k}^s \) for the new scenario \( s \) that results from the incremented weight \( w^*(v_k) = w(v_k) + \delta_{k,1} \). We can similarly compute \( I^s_{R,k} \) from \( I^0_{R,k} \) in constant time. Carrying out these operations whenever a newly expanded cluster is created thus takes \( O(n) \) time for a given \( k \) and \( O(n^2) \) time in total for all \( k \)’s. Define \( \Delta_{R,k} \equiv \bigcup_{k=2}^n \Delta_{R,k} \).

Lemma 8. Assume that \( \Delta_{R,k} \), as well as all the data mentioned in Corollary 6, are available. Then under any given scenario \( s \in S^*_{L,k} \), we can compute the following in \( O(\log n) \) time.

(a) \( \Phi^s(v_i) = \Phi^s_L(v_i) + \Phi^s_R(v_i) \) for any given index \( i \).

(b) \( \Phi^s(x) = \Phi^s_L(x) + \Phi^s_R(x) \) for any given point \( x \).

Among the items in \( \Delta_{R,k} \), there is a natural lexicographical order, ordered first by \( b \) and then by \( w(v_k) \), from the smallest to the largest. We write \( s \prec s' \) if \( s \) is ordered before \( s' \) in this order. In what follows we assume the items in \( \Delta_{R,k} \) are sorted by \( \prec \).

4.2 Tracking sink \( \mu^s \)

Observe that we have \( \Phi^s_L(x) = \Phi^{s,M}_L(x) \) for \( x \leq v_b \), which is independent of \( w(v_b) \). Similarly, we have \( \Phi^s_R(x) = \Phi^{s,M}_R(x) \) for \( x \geq v_b \), which is also independent of \( w(v_b) \). We initialize the current scenario by \( s = s_0 \), the boundary vertex \( v_b \) by \( b = 1 \), and its weight by \( w(v_b) = w^{s_0}(v_1) \).

For each successive increment in \( \Delta_{R,k} \), from the smallest (according to \( \prec \)), we want to know the leftmost 1-sink under the corresponding scenario. It is possible that, as we increase the weight \( w(v_b) \), the sink may jump across \( v_b \) from its right side to its left side, and vice versa, back and forth many times. We shall see how this can happen below.

By Lemma 8, for a given index \( b \), we can compute \( \{ \Phi^{s,i-1}(v_i) \mid i=1,2,\ldots,n \} \) in \( O(n \log n) \) time.\(^5\) We first scan \( \Phi^{s,1}(v_b), \Phi^{s,1}(v_{b-1}), \ldots, \Phi^{s,1}(v_1) \), and whenever we encounter a value smaller than those we examined so far, we record the index of the corresponding vertex. Let \( I^b_L \) be the recorded index set, starting with \( b \). We then scan \( \Phi^{s,2}(v_b), \Phi^{s,2}(v_{b+1}), \ldots, \Phi^{s,2}(v_{n-1}) \) similarly, and let \( I^b_R \) be the recorded index set, starting with \( b \). We now plot point \( (v_i, \Phi^{s,1}(v_i)) \) for \( i \in I^b_L \cup I^b_R \) in the \( x-y \) coordinate system, with distance \( d(v_i, v_1) \) as the \( x \) value and \( \Phi^{s,1}(v_i) \) as the \( y \) value. See Fig. 2, where \( d(v_i, v_1) \) is indicated by \( v_i \). It is clear from the definition that for \( i,j \in I^b_L \) such that \( i < j \), we have \( \Phi^{s,1}(v_i) < \Phi^{s,1}(v_j) \), and for \( i,j \in I^b_R \) such that \( i < j \), we have \( \Phi^{s,1}(v_i) > \Phi^{s,1}(v_j) \). Therefore, the points plotted on the left (resp. right) side of \( v_b \) get higher and higher as we approach \( v_b \) from left (resp. right), as seen by the black dots in Fig. 2.

Note that for a vertex \( v_i \) (\( \prec v_b \)), as \( w(v_b) \) is increased, \( \Phi^s_R(v_i) \) increases, while \( \Phi^s_L(v_i) \) remains fixed at \( \Phi^{s,M}_L(v_i) \), where \( s \) is the scenario reflecting the change in \( w(v_b) \). For \( v_i \prec v_b \), on the other hand, as \( w(v_b) \) is increased, \( \Phi^s_L(v_i) \) increases, while \( \Phi^s_R(v_i) \) remains fixed at

\(^5\) Recall the definition of \( \tau_j \) from Sec. 2.1.
\( \Phi^R(v_i) \). A vertical arrow in Fig. 2 indicates the relative amount of increase in the cost at the corresponding vertex when \( w(v_b) \) is increased by a certain amount. Note that the farther away a vertex is from \( v_b \), the more is the increase in the cost.

The following proposition summarizes the above observations.

**Proposition 9.**
(a) \( \Phi^*(v_i) < \Phi^*(v_j) \) holds for any pair \( i, j \in I^b_L \) such that \( i < j \).
(b) \( \Phi^*(v_i) > \Phi^*(v_j) \) holds for any pair \( i, j \in I^b_R \) such that \( i < j \).
(c) Either the vertex with the smallest index in \( I^b_L \) or the vertex with the largest index in \( I^b_R \) has the lowest cost, i.e., it is a 1-sink.

Note that the cost at \( v_b, \Phi^R(v_b) \), is the highest among the points plotted, and is not affected by the change in \( w(v_b) \). We consider the three properties in Proposition 9 as invariant properties, and remove the vertices that do not satisfy (a) or (b), as we increase \( w(v_b) \). As we increase \( w(v_b) \), in the order of the sorted increments in \( \Delta_R \), we update \( I^b_L \) and \( I^b_R \), looking for the change of the sink.

**Proposition 10.** As \( w(v_b) \) is increased, there is a sink at the same vertex for all the increments tested since the last time the sink moved, until the smallest index in \( I^b_L \) or the largest index in \( I^b_R \) changes, causing the sink to move again. The sink cannot move away from \( v_b \).

We are thus interested in how \( I^b_L \) and \( I^b_R \) change, in particular, when its smallest index in \( I^b_L \) or the largest index in \( I^b_R \) changes.

**Lemma 11.** Let \( i \) and \( j \) be vertex indices such that either they are adjacent in \( I^b_L \) and \( i < j \) holds, or adjacent in \( I^b_R \) and \( i > j \) holds. The smallest\(^6 \) \((b, \delta) \in \Delta_R \), if any, such that increasing \( w(v_b) \) by \( \delta \) above \( w(v_b) \) causes the cost at \( v_i \) to reach or exceed that at \( v_j \), can be determined in \( O(\log^2 n) \) time.

**Proof.** Use binary search on \( \Delta_R \) (sorted by \( \prec \)), and compare the costs at \( v_i \) and \( v_j \) for each probe in \( O(\log n) \) time, using Lemma 8.

\[ \text{If such a } \delta \text{ in Lemma 11 does not exist, we set } \delta = \infty. \] From Lemma 11, it follows that the total time for all adjacent pairs is \( O(n \log^2 n) \). We insert a triple \((\delta; i, j)\) into a min-heap \( \mathcal{H}_b \), organized according to the first component \( \delta \), from which we can extract the item with the smallest first component. For a given \( b \), once \( \mathcal{H}_b \) has been constructed this way, we pop the item \((\delta; i, j)\) with the smallest \( \delta \) from \( \mathcal{H}_b \) in constant time. If \( i, j \in I^b_L \) (resp. \( i, j \in I^b_R \) then

\[ \text{According to } \prec. \]
we remove $i$ (resp. $j$) from $I^b_L$ (resp. $I^b_R$), and compute $(\delta'; i^-, j)$ (resp. $(\delta'; i, j^+)$) where $i^-$ (resp. $j^+$) is the index in $I^b_L$ (resp. $I^b_R$) that is immediately before (resp. after) $i$ (resp. $j$). By Lemma 11 we can find $\delta'$ in $O(\log^2 n)$ time, and insert $(\delta'; i^-, j)$ (resp. $(\delta'; i, j^+)$) into $H_b$, taking $O(\log n)$ time. If $i$ was the smallest index in $I^b_L$, the sink may have moved. In this case no new item is inserted into $H_b$. Similarly, if $j$ was the largest index in $I^b_R$, the sink may have moved, and no new item is inserted into $H_b$.

We repeat this until either $H_b$ becomes empty or the minimum $\delta$-value in $H_b$ is $\infty$. It is repeated $O(n)$ times, so the total time required is $O(n \log^2 n)$. If the sink moves when the smallest index in $I^b_L$ or the largest index in $I^b_R$ changes, we have determined the sink under all the scenarios with the lighter $w(v_b)$ since the last time the sink moved. Once $w(v_b) = w(v_b) + \delta$ reaches $\mathcal{W}(v_b)$, $b$ is incremented, and the new boundary vertex now lies to the right of the old boundary vertex $v_b$ in Fig. 2. For each $b = 1, 2, \ldots, n$, let $S_b = \{s \in S^* | b(s) = b\}$.\footnote{The above method is presented as a formal algorithm in [3].}

Lemma 12.
(a) Sinks $\{\mu^s | s \in S_b \cap S^*_L\}$ can be computed in $O(n \log^2 n)$ time for a given boundary vertex $v_b$.
(b) Sinks $\{\mu^s | s \in S_b \cap S^*_R\}$ can be computed in $O(n \log^2 n)$ time for a given boundary vertex $v_b$.

For the clusters in $C^*_b$ that lie to the right of $C^*_R(v_b)$ and are not merged as a result of an increase in $w(v_b)$, the sum of their intra costs was already precomputed. Repeating the above operations for $b = 1, 2, \ldots, n$, we get our first major result.

Lemma 13. The sinks $\{\mu^s | s \in S^*\}$ can be computed in $O(n^2 \log^2 n)$ time.

5 Minmax regret sink

Now that we know how to compute the sinks $\{\mu^s | s \in S^*\}$, we proceed to compute the upper envelope for the $O(n^2)$ regret functions $\{R^s(x) = \Phi^s(x) - \Phi^s(\mu^s) | s \in S^*\}$. The minmax regret sink $\mu^*$ is at the lowest point of this upper envelope.

5.1 Upper envelope for $\{R^s(x) | s \in S^*\}$

If we try to find the upper envelope $\max_{s \in S^*} \Phi^s(x)$ in a naive way, it would take at least $O(n^3)$ time, since $|S^*| = O(n^2)$, and for each $s$, $\Phi^s(x)$ consists of $O(n)$ linear segments. We employ the following two-phase approach.

Phase 1: For each $b$, compute the upper envelope $\max_{s \in S_b} R^s(x)$.

Phase 2: Compute the upper envelope for the results from Phase 1.

In Phase 1, we successively update the upper envelope, incorporating regret functions one at a time, which can be done in amortized $O(\log^2 n)$ time per regret function. Thus the total time for a given $b$ is $O(n \log^2 n)$ and the total time for all $b$ is $O(n^2 \log^2 n)$. In Phase 2, we then compute the upper envelope for the resulting $O(n)$ regret functions with a total of $O(n^2)$ linear segments in $O(n^2 \log n)$ time. To implement Phase 1, we first present the following lemma.

Lemma 14. Let $s, s' \in S_b$ be two scenarios such that and $s \preceq s'$.
As $x$ moves to the right, the difference $D(x) = \Phi^{s'}(x) - \Phi^s(x)$ decreases monotonically for $v_1 \preceq x \preceq v_b$ and increases monotonically for $v_b \preceq x \preceq v_n$.
We can compute the upper envelope \( R^*(x) \) for the left set and right set separately. Since the second term in (11) is independent of position \( x \), Lemma 14 implies

\[ R^{s'}(x) - R^*(x) = \Phi^{s'}(x) - \Phi^*(x) - \{\Phi^{s'}(x) - \Phi^*(x)\} \]
\[ = \Phi^{s'}(x) - \Phi^*(x) - \{\Phi^{s'}(\mu^*) - \Phi^*(\mu^*)\}. \]  

(11)

Since the second term in (11) is independent of position \( x \), Lemma 14 implies

\[ \text{Lemma 15. Let } s, s' \in S_b \text{ be two scenarios such that } s < s'. \text{ Then } R^{s'}(x) \text{ may cross } R^s(x) \text{ at most once in the interval } [v_b, v_n] \text{ from below.} \]

See Fig. 3 for an illustration for Lemma 15. For \( x > v_b \), we compute \( \max_{s \in S_b} R^s(x) \), updating a partially computed upper envelope \( U(x) \) by successively incorporating the “next” regret function \( R^s(x) \) to it. We can use binary search to find the crossing point of \( U(x) \) and \( R^s(x) \), and invoke Lemma 8.

\[ \text{Lemma 16.} \]
\[ \begin{align*}
\text{(a) The upper envelope } \max_{s \in S_b} R^s(x) \text{ has } O(|S_b| + n) \text{ line segments.} \\
\text{(b) We can compute the upper envelope } \max_{s \in S_b} R^s(x) \text{ in } O(|S_b| \log^2 n) \text{ time.}
\end{align*} \]

\[ \text{Proof. (a) Without loss of generality, consider the upper envelope in the interval } [v_b, v_n]. \text{ Since } R^s(x) = \Phi^s(x) - \Phi^*(\mu^*), R^s(x) \text{ is linear over the edge connecting any adjacent pair of vertices, and } \max_{s \in S_b} \Phi^s(x) \text{ has } O(|S_b| + n) \text{ line segments on all edges by Lemma 15.} \]

\[ \text{(b) See the analysis of Algorithm 3 in [3].} \]

5.2 Main theorem

Since \( \bigcup_{b=1}^n S_b = S^* \) and \( |S^*| = O(n^2) \), Lemma 16 implies that it takes \( O(n^2 \log^2 n) \) time to compute \( \{\max_{s \in S_b} R^s(x) | b = 1, \ldots, n\} \). These \( n \) upper enveopole together have \( O(n^2) \) linear segments. Hershberger [9] showed that the upper envelope of \( m \) line segments can be computed in \( O(m \log m) \) time. We can use his method to compute the global upper envelope for \( \{\max_{s \in S_b} R^s(x) | b = 1, \ldots, n\} \) in \( O(n^2 \log^2 n) \) additional time.

\[ \text{Lemma 17. The upper envelope } \max_{s \in S^*} R^s(x) \text{ can be computed in } O(n^2 \log^2 n) \text{ time.} \]

So far we have paid no attention to the negative spikes in \( R^s(x) \) at vertices. Divide the problem in two subproblems: minmax regret sink is (i) on an edge, and (ii) at a vertex. Compare the two solutions and pick the one with the smaller cost. In addition to Lemma 17, we should evaluate the regret at each vertex. The true minmax regret sink is at the point with the minimum of these maximum regrets. Corollary 6 and Lemmas 3, 13 and 17 imply our main result.
Theorem 18. The minmax regret sink on path networks can be computed in $O(n^2 \log^2 n)$ time.

6 Conclusion

We presented an $O(n^2 \log^2 n)$ time algorithm for finding a minmax regret aggregate time (a.k.a. minsum) sink on path networks with uniform edge capacities, which improves upon the previously most efficient $O(n^3)$ time algorithm in [11]. We hope some methods we devised in this paper will find applications in solving some other related problems. Future research topics include efficiently solving the minmax regret problem for aggregate time sink for more general networks such as trees. No such polynomial time algorithm is known at present.

References


Computing Optimal Shortcuts for Networks

Delia Garijo
Departamento de Matemática Aplicada I, Universidad de Sevilla, Spain
dgarijo@us.es

Alberto Márquez
Departamento de Matemática Aplicada I, Universidad de Sevilla, Spain
almar@us.es

Natalia Rodríguez
Departamento de Matemática Aplicada I, Universidad de Sevilla, Spain
nrodriguez@dc.uba.ar

Rodrigo I. Silveira
Departamento de Matemàtiques, Universitat Politècnica de Catalunya, Spain
rodrigo.silveira@upc.edu

Abstract
We study augmenting a plane Euclidean network with a segment, called shortcut, to minimize the largest distance between any two points along the edges of the resulting network. Questions of this type have received considerable attention recently, mostly for discrete variants of the problem. We study a fully continuous setting, where all points on the network and the inserted segment must be taken into account. We present the first results on the computation of optimal shortcuts for general networks in this model, together with several results for networks that are paths, restricted to two types of shortcuts: shortcuts with a fixed orientation and simple shortcuts.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases graph augmentation, shortcut, diameter, geometric graph

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.15


Funding D. G. and R. S. were supported by project MTM2015-63791-R. R. S. was also supported by Gen. Cat. 2017SGR1640 and MINECO through the Ramón y Cajal program. A. M. was supported by project BFU2016-74975-P.

1 Introduction

A fundamental task in network analysis, especially in the context of geographic data (for instance, for networks that model roads, rivers, or train tracks), is analyzing how an existing network can be improved. This can arise in many different contexts: in relation to facility location analysis, for instance, to guarantee a certain maximum travel time from any point on the network to the nearest hospital, or in road network design problems, to decide where to add road segments to reduce network congestion [16].

Networks like the ones above are naturally modeled as a geometric network: an undirected graph whose vertices are points in R^2 and whose edges are straight-line segments connecting pairs of points. Moreover, in many applications, it is reasonable to assign lengths to the edges equal to the Euclidean distance between their endpoints. These are called Euclidean
networks. When, in addition, there are no crossings between edges, the Euclidean network is said to be plane. Many problems in geographic analysis, for instance, those involving transportation networks, can be accurately modeled with a plane Euclidean network. In the following, we shall simply write network, it being understood as plane and Euclidean.

One of the most fundamental ways to improve a network is by adding edges. This increases the connectivity of the network and potentially can decrease travel times and congestion. The most studied criteria to measure network improvement, in the geometric setting, are related to distances. Particularly important is the maximum distance, or diameter of the network, which provides an upper bound on the distance between any two network points. Another important distance-related criterion in this context is the dilation, which captures the maximum detour between two points on the network.

In this work, we focus on the problem of adding edges to a network in order to improve its diameter. This can be seen as a variant of the Diameter-Optimal-$k$-Augmentation problem, which consists in inserting $k$ additional segments into a graph, while minimizing the largest distance in the resulting network (see the survey [14] for more on augmentation problems over plane geometric graphs). More precisely, we study a continuous version of the problem for $k = 1$: we consider the addition of one segment, called shortcut, whose endpoints can be any two points (not necessarily vertices) on the network. A segment will be considered a shortcut only if its insertion improves the diameter of the resulting network. Note that the resulting network includes the points on the shortcut inserted.

Our goal is to find an optimal shortcut: one minimizing the diameter of the resulting network, over all possible shortcuts.

Two major variants of the problem arise, depending on how the shortcut is inserted into the network. In the first variant, which we call highway model, the crossings between the shortcut and the network edges do not form new network vertices: a path can only enter and leave the shortcut through its endpoints. In contrast, in the planar model, every crossing creates a new vertex, which can be used by paths in the network. Figure 1 illustrates some of the differences between the two models.

In this work, we focus on the planar model. This model is more general, and is applicable to a wider range of situations, like the addition of segments to road or pedestrian networks. From a theoretical point of view, the difference between the highway and planar model is important. The latter results in more complex problems, since the fact that a shortcut can be used only in part, implies that the structural information on how the distances in the network change after adding a segment is more difficult to maintain. Moreover, as we show in this work, many intuitive properties of shortcuts do not hold in the planar model anymore.
Related work. There has been a considerable amount of work devoted to the graph version of the Optimal-k-Augmentation problem. Due to space constraints, we only discuss the geometric version of the problem (i.e., where the graph is embedded in the plane), and where the continuous diameter is used (i.e., the distances are taken over all pairs of points in the network, as opposed to considering only pairs of vertices).

Most attention to the problem studied here has been on the highway model, for certain classes of graphs. For paths, De Carufel et al. [6] gave an algorithm to find an optimal shortcut in linear time, and also optimal pairs of shortcuts (i.e., \(k = 2\)) for convex cycles.

Trees have been studied in a recent follow-up work [7], which presents an algorithm to find a simple shortcut, a variant that has been studied before, which has applications in settings generalizing an existing result for paths [15]. Section 3 focuses on paths: we first show that algorithms to compute an optimal shortcut if one exists. Moreover, we present a discretization of the problem that immediately leads to an approximation algorithm for general networks, generalizing an existing result for paths [15]. Section 3 focuses on paths: we first show that the diameter of a path network after adding a shortcut can be computed in \(\Theta(n)\) time. Then we improve the method of Section 2 for shortcuts of any fixed direction. Finally, we study simple shortcuts, a variant that has been studied before, which has applications in settings where the added edge cannot intersect the existing network.

Due to space limitations, most proofs are not included here; they can be found in [12].

1.1 Preliminaries

We will use \(\mathcal{N} = (V(\mathcal{N}), E(\mathcal{N}))\) to denote a network with \(n\) vertices, and \(\mathcal{N}_f\) for its locus, the set of all points of the Euclidean plane that are on \(\mathcal{N}\). Thus, \(\mathcal{N}_f\) is treated indistinctly as a network or as a closed point set. When \(\mathcal{N}_f\) is a path, we use \(P_f\) instead of \(\mathcal{N}_f\). Further, we write \(a \in \mathcal{N}_f\) for a point \(a\) on \(\mathcal{N}_f\), and \(V(\mathcal{N}) \subseteq \mathcal{N}_f\).

A path \(P\) connecting two points \(a, b\) on \(\mathcal{N}_f\) is a sequence \(a u_1 \ldots u_kb\) such that \(u_1u_2, \ldots, u_{k-1}u_k \in E(\mathcal{N})\), \(a\) is a point on an edge (\(\neq u_1u_2\)) incident to \(u_1\), and \(b\) is a point on an edge (\(\neq u_{k-1}u_k\)) incident to \(u_k\). We use \(|P|\) to denote the length of \(P\), i.e., the sum of the lengths of all edges \(u_1u_2, \ldots, u_{k-1}u_k\) plus the lengths of the segments \(au_1, bu_k\). The length of a shortest path from \(a\) to \(b\) is the distance between \(a\) and \(b\) on \(\mathcal{N}_f\). This distance is written as \(d_{\mathcal{N}_f}(a, b)\) or \(d(a, b)\) when the network is clear, and whenever \(ab \notin E(\mathcal{N}_f)\), it is larger than \(|ab|\), the Euclidean distance between the points.

The eccentricity of a point \(a \in \mathcal{N}_f\) is \(\text{ecc}(a) = \max_{b \in \mathcal{N}_f} d(a, b)\), and the diameter of \(\mathcal{N}_f\) is \(\text{diam}(\mathcal{N}_f) = \max_{a \in \mathcal{N}_f} \text{ecc}(a)\). Two points \(a, b \in \mathcal{N}_f\) are diametral whenever \(d(a, b) = d(\mathcal{N}_f)\), and a shortest path connecting \(a\) and \(b\) is then called a diametral path.

The diameter of \(\mathcal{N}_f\), \(\text{diam}(\mathcal{N}_f)\), can be computed in polynomial time [5, 8]. Furthermore, the diametral pairs of \(\mathcal{N}_f\) are either (i) two vertices, (ii) two points on distinct non-pendant
edges, or (iii) a pendant vertex and a point on a non-pendant edge [5, Lemma 6]. Thus, with some abuse of notation, in Section 2, we will say that a \textit{diametral pair} $\alpha, \beta \in V(N) \cup E(N)$ may be (i) vertex-vertex, (ii) edge-edge, or (iii) vertex-edge.

A \textit{shortcut} for $N_f$ is a segment $s$ with endpoints on $N_f$ such that $\text{diam}(N_f \cup s) < \text{diam}(N_f)$. We say that shortcut $s$ is \textit{simple} if its two endpoints are the only intersection points with $N_f$, and $s$ is \textit{maximal} if it is the intersection of a line and $(N_f \cup s)$, i.e., $s = (N_f \cup s) \cap \ell$, for some line $\ell$. A shortcut is \textit{optimal} if it minimizes $\text{diam}(N_f \cup s)$ among all shortcuts $s$ for $N_f$.

### 2 General networks

The main result in [5] states that one can always determine in polynomial time whether a network $N_f$ has a shortcut (and compute one, in case of existence). In this section, we first prove the analogous result for optimal shortcuts. Our proof uses some ideas in [5] but captures the property of being optimal with a much shorter argument based on some functions defined in Lemma 1 below.

Let $\alpha, \beta \in V(N) \cup E(N)$, and let $e = uv$ and $e' = u'v'$ be two edges of $N$. When $\alpha$ is an edge, we use $\text{ecc}(u, \alpha)$ to indicate the maximum distance from $u$ to the points on $\alpha$ (analogous for $\beta$ and the remaining endpoints of $e$ and $e'$); if $\alpha$ is a vertex, $\text{ecc}(u, \alpha) = d(u, \alpha)$. In general, $\text{ecc}(\alpha, \beta) = \max_{t \in \alpha, z \in \beta} d(t, z)$.

\begin{lemma}
Let $y = ax + b$ be a line intersecting edges $e = uv$ and $e' = u'v'$ on points $p$ and $q$, respectively, and let $\alpha, \beta \in V(N) \cup E(N)$. For each pair $(w, z)$ with $w \in \{u, v\}$ and $z \in \{u', v'\}$, function $f_{\alpha,\beta}^{w,z}(a, b) = \text{ecc}(w, \alpha) + |wp| + |pq| + |qz| + \text{ecc}(z, \beta)$ is linear in $b$.
\end{lemma}

The following theorem is the optimality version of Theorem 8 in [5].

\begin{theorem}
It is possible to determine in polynomial time whether a network $N_f$ admits an optimal shortcut, and compute one in case of existence.
\end{theorem}

It should be noted that the approach that leads to the preceding result, albeit polynomial, has a very high running time. A direct implementation involves $O(n^4)$ functions $f_{\alpha,\beta}^{w,z}(a, b)$ that must be computed, and this has to be done for $O(n^4)$ different cases. Moreover, each evaluation of $f_{\alpha,\beta}^{w,z}(a, b)$ takes $O(n^2)$ time. All in all, its running time would add up to $O(n^{10})$.

### 2.1 Discretizing the set of possible shortcuts: approximation

In light of the high running time of the previous approach, it becomes interesting to look for faster approximation algorithms. Moreover, given the continuous nature of the problem, it is natural to wonder to what extent the problem can be discretized. In other words, how good can shortcuts be if we restrict them to some discrete collection of segments? The most natural choice for such a collection is probably the segments defined by pairs of vertices $u, v$ of $N_f$, but this choice can lead to poor results, as the example in Figure 2(left) shows. In some cases, one can do better by considering the \textit{maximal extensions} of the segments $uv$ (i.e., the largest segment through $uv$ with endpoints on $N_f$), as Yang [15] did to obtain an additive approximation for paths. Unfortunately, as Figure 2(right) shows, maximal extensions do not work anymore as soon as $N_f$ is a tree. However, in this section, we show that if one considers all extensions of segments defined by two vertices of $N_f$, then it is possible to guarantee an approximation factor for general networks.

\begin{figure}[h]  
\centering  
\includegraphics[width=\textwidth]{figure2.png}  
\caption{Discretizing the set of possible shortcuts: approximation}
\end{figure}

\footnote{An edge $uv \in E(N)$ is pendant if either $u$ or $v$ is a pendant vertex (i.e., has degree 1).}
Figure 2 Left: the optimal shortcut is the dashed purple segment, which contains the blue segment. That blue segment (and any other segment between two vertices) gives a larger diameter as points $a$ and $b$ are diametral for both segments. Right: the original diameter is given by the orange path. The best shortcut connecting two vertices is $bc$. Contrary to intuition, extending $bc$ to $bd$ worsens the diameter, which becomes given by points $a$ and $e$ (pink path).

Figure 3 Left: approximating a shortcut $s$ with a segment $s' \in S_2$. Right: using $s'$ instead $s$ to go from $a$ to $b$ causes a detour of at most $4\rho$ (purple path).

Let $S$ be the infinite set of segments with endpoints in $\mathcal{N}_t$, and let $S_2 \subset S$ be the subset of segments of $S$ that contain two vertices of $\mathcal{N}_t$. The following proposition states that set $S_2$ is an approximation of $S$.

Proposition 3. Let $\rho$ be largest edge length in $\mathcal{N}_t$. Then, $\min_{s \in S} \diam(\mathcal{N}_t \cup s) \leq \min_{s \in S_2} \diam(\mathcal{N}_t \cup s) \leq \min_{s \in S} \diam(\mathcal{N}_t \cup s) + 4\rho$.

Proof. The first inequality is straightforward. For the second, it suffices to prove that given $s = pq \in S \setminus S_2$ there exists $s' \in S_2$ such that $\diam(\mathcal{N}_t \cup s') \leq \diam(\mathcal{N}_t \cup s) + 4\rho$.

Segment $s$ may cross several faces of $\mathcal{N}_t$, refer to Figure 3. Consider the first and the last ones, say $F_1$ and $F_2$, together with the vertices of $\mathcal{N}_t$ that are adjacent to $p$ and $q$ in those faces: $u, v$ in $F_1$ and $u', v'$ in $F_2$. Let $V_1$ be the vertices of $\mathcal{N}_t$ in the quadrilateral $upqu'$ (including $u$ and $u'$), and let $C_1$ be its convex hull. Analogously, we have $V_2$ and $C_2$ for the quadrilateral $upqv'$. Note that both convex hulls may have one point in common. Extending one of the common internal tangents of $C_1$ and $C_2$ gives rise to a segment $s'$ with endpoints on two of the edges of $F_1$ and $F_2$ containing points $p$ and $q$. Observe that $s'$ intersects all the edges of $\mathcal{N}_t$ that are crossed by $s$. Thus, this construction allows us to show that, for any two points $a, b \in \mathcal{N}_t$, the length of the shortest path between $a$ and $b$ that uses $s'$ is at most $4\rho$ plus the corresponding length but using $s$. To do this, we first use the triangle inequality to compare the lengths of the used portions of segments $s$ and $s'$, which gives a difference of $2\rho$, and then we add the two distances indicated in Figure 3(right). A similar argument is used for $a \in s'$ and $b \in \mathcal{N}_t$.

The collection $S_2$ is finite but quite large, it has size $O(n^4)$, which gives a time complexity of $O(n^6)$ to compute the optimal among the segments in $S_2$. (there are $O(n^2)$ possible extensions per each pair of vertices, and for each of them one needs to compute the diameter from scratch in $O(n^2)$ time [13]).
We would like to find a small subset of $S_2$ that preserves the property in Proposition 3. Ideally, we would like to consider not all the extensions of a segment with endpoints in $V(N_\ell)$ (that is exactly $S_2$), but only the best extension for each segment. Unfortunately, this appears rather difficult: already for a tree with a single vertex of degree larger than two, it may happen that an extension of a segment gives a worse diameter than the segment itself, see Figure 2(right). However, we show next that we can speed-up the computation of the diameter for each extension in $S_2$, saving a nearly-linear factor in the total running time.

Given a segment $s' = p'q'$, let $r$ be the ray starting at $p'$ and containing $s'$, and let $\mathcal{P} = p_0, p_1, \ldots, p_k$ be the sorted list of intersection points of $r$ with edges of $N_\ell$ (note that $q' = p_j$ for some $j$). Segments $s_i = p'p_i$ are called extensions of $s'$ to the right; the extensions to the left are defined similarly. Next we show how to speed-up the re-computation of the diameter of $N_\ell$ as we insert $s_0, s_1, \ldots, s_k$, in that order. To that end, we split the re-computation of distances into two parts: distances from points on $s_i$ to points on $N_\ell$, and distances (in $N_\ell$) between two points on $N_\ell$. 

**Lemma 4.** Let $u$ and $v$ be vertices of $N_\ell$. It is possible to compute the eccentricities of all the extensions to the right of segment $uv$ in $O(n^2)$ time.

**Proof.** As a preprocessing step, we store the distances from each vertex to all the other edges and the point at each edge attaining that maximum distance. This allows us to construct the functions $\Phi_{uv}^{st} : [0, 1] \to \mathbb{R}^+$ that encode the information of the maximum distance from each point on an edge $uv$ to an edge $st$ (see [13, Theorem 2] for their analytic expression). Their shape is as follows – see also Figure 4(left): let $u'$ and $v'$ be the farthest points to, respectively, $u$ and $v$ in edge $st$. Function $\Phi_{uv}^{st}$ increases uniformly from 0 and from 1 until the distance between both lines equals the distance between $u'$ and $v'$, at that moment it stabilizes horizontally. Thus, knowing the farthest points $u'$ and $v'$ to $u$ and $v$ in the segment $st$ (and the distance between them), it is possible to build $\Phi_{uv}^{st}$ in constant time.

The main idea of the proof of this lemma is that it is possible to update each map $\Phi_{uv}^{st}$ for each extension of a segment again in constant time. Observe that $\Phi_{uv}^{st}$ encodes the information of the largest distance from any point of $uv$ to the segment $st$.

In a first step, we insert segment $s_0 = uv$. As $u$ and $v$ are in $N_\ell$, they belong to some edges $g$ and $g'$, and we use the information of $\Phi_{uv}^{st}$ and $\Phi_{uv}^{st'}$ to find the largest distance from $u$ and $v$ to $st$ (in $N_\ell$). With that information, we compute $\Phi_{uv}^{st'}$ in constant time. Thus, the maximum eccentricity of the edge $s_0 = uv$ can be computed in linear time.

Observe that building the map $\Phi_{uv}^{st'}$ it is possible to detect if $ecc(v, st)$ changes when adding $s_0$, so, we update the values of the distances from $vp_0$ to all the other edges, and the point on each edge giving that maximum distance (again, in linear time).
Therefore, we can look for that minimum by binary search, computing with green arrows must be tested (they are the maximal blue points).

Of course, the addition of \( s_0 \) can change the eccentricity of \( p_1 \) with respect to some other edge (and the same with any of the other \( p_i \)'s), but we will see that we do not need to update that information at this moment. Indeed, if the distance from \( p_1 \) to \( st \) changes when adding \( s_0 \), it can only decrease. Then, the addition of \( s_1 \) is going to make it even smaller – see Figure 4(right). Thus, in step \( i \), we only need to update the information of the new vertex \( p_i \), since by adding \( s_i \), the value of \( p_{i+1} \) is going to be updated. ▶

Lemma 5. Let \( u \) and \( v \) be vertices of \( N_\ell \). It is possible to find the extension \( s \) of segment \( uv \) that minimizes \( \text{diam}(N_\ell \cup s) \) in \( O(n^3 \log n) \) time.

Proof. The value of \( \text{diam}(N_\ell \cup s) \) can be computed by calculating the eccentricity of segment \( s \) and comparing with the eccentricities in \( N_\ell \cup s \) of all the points in \( N_\ell \). Thus, for each extension \( s' \) of \( uv \) to the left, we compute the eccentricities \( E(i) \) of all its extensions \( s_i \) to the right using Lemma 4, in \( O(n^2) \) time. Let \( N(i) \) be the maximum distance in \( N_\ell \cup s_i \) between pairs of points in \( N_\ell \). Our goal is to compute \( \min_i \max\{E(i), N(i)\} \). Since \( N(i) \) is a decreasing function as \( i \) grows, we do not need to compute \( N(i) \) for all values of \( i \), but only for those \( i \) for which \( E(i) \) is maximal: there is no \( j > i \) with \( E(j) < E(i) \) (see Figure 5). Therefore, we can look for that minimum by binary search, computing \( N(i) \) only for \( O(\log n) \) values of \( i \). Using [10], we can update the distances between vertices in quadratic time and then compute \( N(i) \) also in quadratic time (the distance between pairs edge–edge and vertex-edge can be computed in constant time knowing the distance between vertices), giving a total time of \( O(n^3 \log n) \). ▶

We thus obtain the main result in this section.

Theorem 6. Let \( \rho \) be the length of a longest edge of a network \( N_\ell \). Then, it is possible to find a segment \( s' \) such that \( \text{diam}(N_\ell \cup s') \leq \min_{s \in \mathcal{S}} \text{diam}(N_\ell \cup s) + 4\rho \) in \( O(n^5 \log n) \) time.

This result immediately gives a simple approximation algorithm: subdivide each edge in \( N_\ell \) by adding dummy vertices such that the largest resulting edge length is \( \epsilon \). Then the previous theorem implies the following result, which is a generalization to general networks of the result for paths presented in [15, Theorem 8.1].

Corollary 7. Let \( \rho \) be the length of a longest edge of a network \( N_\ell \). Then, for any \( 0 < \epsilon < \rho/2 \) it is possible to find a segment \( s' \) such that \( \text{diam}(N_\ell \cup s') \leq \min_{s \in \mathcal{S}} \text{diam}(N_\ell \cup s) + 4\epsilon \) in \( O((n\rho/\epsilon)^5 \log(n\rho)) \) time.
Diameter after inserting a shortcut

The diameter of $P_t$ can be immediately computed in linear time, however, the addition of a shortcut $s$ can create a linear number of new faces, thus in principle it is not clear whether $\text{diam}(P_t \cup s)$ can be computed in linear time, i.e., without computing the diameter between each pair of faces. The main result in this section is that this is still possible.

Path networks have the nice property that the maximal extension of an optimal shortcut is also optimal [15]. Thus, we can assume that $s = pq$ is maximal and horizontal. The insertion of $s$ splits $P_t$ into polygonal chains, which bound the different faces created. Our goal is to compute the pair of chains that have maximum distance in $P_t \cup s$.

We number the polygonal chains from 0 to $m$ in the order of their left endpoints from left to right along $s$ (using right endpoints to disambiguate). Except for possibly the first and last, all chains have both endpoints on $s$. For the $i$th chain $C_i$, we denote its left and right endpoints by $p_i^l$ and $p_i^r$, respectively. If the first vertex of $P_t$ is not on $s$, we consider the path from its first vertex to the first intersection of $P_t$ with $s$ as a degenerate loop chain with equal left and right endpoints on $s$ (analogous for the last vertex of $P_t$). Refer to Figure 7.

Let $|C_i|$ be the length of $C_i$, let $L_i = |pp_i^l|$ and $R_i = |p_i^r q|$, and let $s_i$ denote the segment $p_i^l p_i^r$. Note that $C_i \cup s_i$ forms a cycle. We use $D_i$ for the distance on $P_t \cup s$ from $p_i^l$ to its furthest point $p_i^r$ on $C_i \cup s_i$ (i.e., $D_i$ is the semiperimeter of $C_i \cup s_i$).

We make some basic observations about the diameter between two chains, depending on their relative position. They reveal a key property of the problem: the linear ordering between chains induced by $s$ defines uniquely how the diameter between two chains is achieved.

Observation 8 (Disjoint chains). Let $C_i, C_j$ be two chains of $P_t \cup s$ with $s_i \cap s_j = \emptyset$ and $s_i$ to the left of $s_j$. The diameter of $C_i \cup p_i^l p_j^r \cup C_j$ is $D_i + |p_i^l p_j^r| + D_j = D_i + R_i - R_j - |s_j| + D_j$. 

Figure 6 Schematic construction showing that the insertion of a shortcut $pq$ can create $\Theta(n^2)$ diametral pairs. The distance between the top of one spike on the left of $o$ and one on its right, like $p_i$ and $q_j$, can be made to be $|pq|$, and equal to the diameter of $P_t \cup pq$. 

3 Path networks

In the remaining, we focus on networks that are paths. To illustrate the complexity of this seemingly simple setting, we begin by observing that the insertion of a shortcut to a path can create a quadratic number of diametral pairs; as illustrated in the construction in Figure 6. It consists of $\Theta(n)$ spikes placed symmetrically with respect to the midpoint of the shortcut, denoted with $o$. After inserting $pq$, each spike forms a face with a cycle of length roughly twice its height. The spikes are spaced by one unit each, while their heights are set such that the distance from $o$ to the top of the spike is always the same, namely $|pq|/2$. In this way, for any two spike tops $p_i$ and $q_j$ on the left and right of $o$, respectively, the distance between $p_i$ and $q_j$ on $P_t \cup pq$ is always equal to $|pq|$, which is also the diameter of $P_t \cup pq$. 

3.1 Diameter after inserting a shortcut
The computation of the information in (1) is straightforward when sweeping along \( s \). Within the same running time we can compute \( \text{diam}(\mathcal{P}_k \cup s) \) for Jordan sorting [11] to obtain the intersections in the order along \( s \). For a chain \( C_j \), the position corresponding to its left endpoint has a value equal to \( \beta_j \) in the array \( A_n \), and value \( \gamma_j \) in array \( A_s \). The values corresponding to the right endpoints of the chains are not used, i.e., they have value \(-\infty\), in both arrays. At each array position, we also store pointers to the corresponding chains.

\[ \text{diam}(\mathcal{P}_k \cup s) = \frac{1}{2}(|C_i| + |p_i^l|) \]
To find the nested or overlapping chain furthest from \( C_i \), we would like to perform one maximum range query in \( A_n \) and one in \( A_o \), in both cases with a subarray corresponding to the interval between the endpoints of \( C_i \). The goal is to use these queries to obtain the furthest chain of each type: nested and overlapping. However, there is an issue. In the way \( A_n \) and \( A_o \) are defined, the result of a range query cannot distinguish between nested or overlapping chains, it necessarily searches in both sets (i.e., \( N_i \cup O_i' \)). Fortunately, the geometry of the problem guarantees that we can still use the result obtained, as we show next. The following lemma shows that if the furthest face is associated to a \( \beta_i \) value, then it must be nested, and similarly, if it is associated to a \( \gamma_i \) value, it must be overlapping.

**Lemma 11.** Let \( C_k \) be a chain with distance to \( C_i \) equal to \( d^* = \max \{ \max_{j \in (N_i \cup O_i')} |C_i| - L_i - R_i + \beta_j, \max_{j \in (N_i \cup O_i')} |C_i| - L_i + R_i + \gamma_j \} \). Then it holds: (i) if \( d^* = |C_i| - L_i - R_i + \beta_k \), then \( k \in N_i \); (ii) if \( d^* = |C_i| - L_i + R_i + \gamma_k \), then \( k \in O_i' \).

Therefore, when processing a chain \( C_k \), we perform one maximum range query in \( A_n \) and one in \( A_o \), and keep the maximum of those two values. Lemma 11 guarantees that the associated chain is the furthest one that is either nested or overlapping. Proceeding in an analogous way for the chains that are overlapping with one endpoint to the left of \( C_k \), the furthest face from \( C_k \) of any of the three types (disjoint, nested, overlapping) can be found in \( O(1) \) time, and the maximum distance between two chains can thus be found in linear time.

**Theorem 12.** For every path \( P_t \) with \( n \) vertices and a shortcut \( s \), it is possible to compute the diameter of \( (P_t \cup s) \) in \( \Theta(n) \) time.

It is worth noting that the ideas used in this section do not extend to networks that are trees, since in that case the structural results in Observations 8–10 do not hold anymore.

### 3.2 Optimal horizontal shortcuts

In this section we compute an optimal horizontal shortcut for a path considerably faster than using the general method in Section 2. After a suitable rotation, this allows to find an optimal shortcut of any fixed orientation.

Assume as in Section 3.1 that shortcuts are horizontal and maximal, so they can be treated as horizontal lines. Now, consider the vertices in \( P_t \) sorted increasingly by \( y \)-coordinate, and let \( y_a, y_b \), with \( y_a < y_b \), be the \( y \)-coordinates of two consecutive vertices in that order. Observations 8–10 are stated in terms of chains, but they also apply to faces. Indeed, they imply that the distance between any two faces \( f_i \) and \( f_j \) is a linear function \( d_{ij}(y) \) for \( y_a \leq y \leq y_b \). Thus, each face \( f_i \) is associated with \( k-1 \) lines in 2D where \( k \) is the total number of faces (each line represents the corresponding function \( d_{ij}(y) \) for \( j \neq i \)). Considering all faces, we obtain a set \( \mathcal{L} \) of \( O(k^2) \) lines (note that \( k = O(n) \)). The optimal shortcut over all \( y \in [y_a, y_b] \) is given by the minimum of the upper envelope of \( \mathcal{L} \), which can be computed in \( O(k^2 \log k) \) time [9]. If this is done with each of the \( n-1 \) horizontal strips formed by consecutive vertices of \( \mathcal{N}_t \), the optimal horizontal shortcut is obtained in total \( O(n^3 \log n) \).
time. Now, this method can be improved if, instead of computing from scratch the upper envelope of \( L \) at each horizontal strip, we maintain the upper envelope between consecutive strips and only add or remove the lines that change when going from one strip to the next one. The changes between two consecutive strips are of three types: (i) one of the two line segments bounding a face within the strip changes; (ii) a face ends; (iii) a new face appears.

In the worst case, \( n - 1 \) lines are removed from \( L \) and another \( n - 1 \) lines are added to \( L \). Maintaining the upper envelope of \( N \) lines is equivalent to maintaining the convex hull of \( N \) points in 2D, which can be done in amortized \( O(\log N) \) time per insert/delete operation with a data structure of size \( O(N) \) \cite{4}. Since we have \( N = O(n^2) \), we obtain the following:

\[ \text{Theorem 13.} \quad \text{For every path } P_t \text{ with } n \text{ vertices, it is possible to find an optimal horizontal shortcut in } O(n^2 \log n) \text{ time, using } O(n^2) \text{ space.} \]

### 3.3 Optimal simple shortcuts

In this section we consider optimal simple shortcuts, i.e., we restrict the possible shortcuts to those whose interior does not intersect \( N_t \). We show that an optimal simple shortcut can be computed much faster if it exists. Note that one must distinguish between an optimal simple shortcut and a simple optimal shortcut. The first is a shortcut that is optimal in the set of simple shortcuts; this is different of being optimal in the set of all shortcuts and, in addition, to be simple. Interestingly, it is known that optimal simple shortcuts may not exist, even for paths \cite{16} (e.g., when the only optimal shortcut goes through a vertex, see \cite[Figure 12(a)]{12}). It is not clear, however, what the conditions for a network \( N_t \) to have an optimal shortcut are, even restricted to simple shortcuts. The following proposition is a first approach to this question (note that its converse is not true, see \cite[Figure 12(b)]{12}).

\[ \text{Proposition 14.} \quad \text{Let } N \text{ be a network whose locus } N_t \text{ admits a simple shortcut, and let } \overline{N} \text{ be the network resulting from adding to } N \text{ all edges of the convex hull of } V(N). \text{ If all faces of } \overline{N} \text{ are convex, then } N_t \text{ has an optimal simple shortcut.} \]

We now turn our attention to the computation of an optimal simple shortcut if one exists.

Let \( s = pq \) be a simple shortcut for a path \( P_t \) with endpoints \( u, v \). Suppose that point \( p \) is closer to \( u \) than \( q \) along \( P_t \); let \( x = d(u, p) \) and \( y = d(v, q) \). There is only one bounded face in \( P_t \cup s \) whose boundary is a cycle \( C(p, q) \). Let \( \overline{p} \) and \( \overline{q} \) be the farthest points from, respectively, \( p \) and \( q \) on \( C(p, q) \), and let \( z = (d(\overline{p}, p) - |pq|)/2 \). Note that \( d(\overline{p}, \overline{q}) = |pq| \) and \( z = d(p, \overline{q}) = d(\overline{p}, q) \). There are three candidates for diametral path in \( P_t \cup s \) (see \cite{6}):

1. The path from \( u \) to \( v \) via \( s \) is diametral if and only if \( z = \min\{x, y, z\} \).
2. the path from \( u \) to \( \overline{p} \) via \( s \) is diametral if and only if \( y = \min\{x, y, z\} \).
3. the path from \( v \) to \( \overline{q} \) via \( s \) is diametral if and only if \( x = \min\{x, y, z\} \).

Thus, \( \text{diam}(P_t \cup s) \in \{x + y + |pq|, x + z + |pq|, y + z + |pq|\} \). For the highway model, it was proved in \cite{6} that \( P_t \) has an optimal shortcut satisfying \( x = y \), which allows to compute it in linear time. In the planar model the situation is more complicated but, in a similar fashion, we can prove the following lemma, which lead to Theorem 16.

\[ \text{Lemma 15.} \quad \text{Let } pq \text{ be an optimal simple shortcut for } P_t. \text{ The following statements hold.} \]

1. If neither \( p \) nor \( q \) are vertices of \( P_t \) then \( x = y = z \).
2. If \( p \) or \( q \) are vertices of \( P_t \) then the two smallest values among \( x, y, z \) are equal.

\[ \text{Theorem 16.} \quad \text{It is possible to decide whether a path } P_t \text{ with } n \text{ vertices has an optimal simple shortcut and compute one (in case of existence) in } O(n^2) \text{ time.} \]
Conclusions

In this work, we have presented the first results on the computation of optimal shortcuts for the planar model. We have shown that an optimal shortcut can be computed in polynomial time, and given a discretization of the problem that results in an approximation of the original continuous version. Even though the discretization obtained is too large to be of practical use, it is interesting from a theoretical point of view, and hopefully will be useful to obtain smaller discretizations in the future. We also presented new results for paths, including how to quickly compute the diameter after inserting a shortcut, the computation of an optimal shortcut of fixed orientation, and of an optimal simple shortcut. These are important first steps on a relevant and difficult problem, which leave many intriguing questions open. The existence of small discrete set of segments to approximate an optimal shortcut, or a fast algorithm to find an optimal shortcut for paths (any orientation), are some examples. Finally, the questions studied but for optimal sets of \( k > 1 \) shortcuts pose challenging open problems.

References

Algorithmic Channel Design

Georgia Avarikioti
ETH Zurich, Switzerland
zetavar@ethz.ch

Yuyi Wang
ETH Zurich, Switzerland
yuwang@ethz.ch

Roger Wattenhofer
ETH Zurich, Switzerland
wattenhofer@ethz.ch

Abstract
Payment networks, also known as channels, are a most promising solution to the throughput problem of cryptocurrencies. In this paper we study the design of capital-efficient payment networks, offline as well as online variants. We want to know how to compute an efficient payment network topology, how capital should be assigned to the individual edges, and how to decide which transactions to accept. Towards this end, we present a flurry of interesting results, basic but generally applicable insights on the one hand, and hardness results and approximation algorithms on the other hand.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases blockchain, payment channels, layer 2 solution, network design, payment hubs, routing

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.16


1 Introduction

Cryptocurrencies such as Bitcoin [15] or Ethereum [1] have a serious throughput problem [6]. They can process tens of transactions per second, whereas non-blockchain systems (credit card companies, inter-banking payment systems, paypal, etc.) can handle tens of thousands of transactions per second. Various proposals have been made in an attempt to solve this throughput problem, e.g., sharding [13, 12] or sidechains [4]. However, payment networks (also known as channels) [7, 16, 2] are widely accepted to be the most promising of these so-called “layer 2” solutions, since payment networks allow data to go off-chain securely.

Duplex micropayment channels [7], Lightning [16] or Raiden [2] are fast and scalable payment networks, where transactions between two users are executed in off-chain two-party channels. The blockchain is involved when opening a channel, as the foundation of a channel must be registered with the blockchain. In exceptions, if the two parties of a channel are in disagreement, the blockchain may also be involved as a safety net when closing a channel.

While the efficiency of channels is undisputed, payment networks have a reputation to be capital hungry and as such difficult to deploy. In this paper we want to better understand this demand for capital, studying the issue from an algorithmic perspective. We want to know the complexity an operator of a payment network, a Payment Service Provider (PSP), will face when setting up a payment network.
1.1 From Payment Channels to Network Design

Consider a PSP wants to create a payment network. The PSP can open a channel between any two parties; technically this can be achieved using multi-party channels [5], where the two parties and the PSP join a three-party channel funded only by the PSP.

Algorithmically speaking, a payment network is a graph, where each undirected edge \((u, v)\) is a payment channel between the parties \(u, v\). When a channel (an edge) is established, PSP capital is locked into the channel on each side of the edge. This capital can then be moved on the channel, from \(u\) to \(v\) or vice versa, much like moving tokens from one side of an abacus to the other. For example, if initially a capital of 5 is locked on each side of the \((u, v)\) channel, then a transaction with a value of 2 from \(u\) to \(v\) will reduce the capital on \(u\)’s side to 3, and increase the capital on \(v\)’s side to 7. Transactions can also be multi-hop, moving capital on each edge of the path, in the direction of the path of the transaction. The only constraint is that the capital on any side of any edge must be non-negative at all times.

The PSP needs to decide how to design the network, i.e., which edges (channels) the PSP should establish. Moreover, the PSP needs to decide how much capital it should assign to these newly established edges, in particular how much capital on each side of every edge.

Establishing a new channel not only involves capital (which is going to be reclaimed eventually), but will also cost (since each newly established channel needs to be registered with the blockchain). We model this channel opening cost as a constant, given that the fee the blockchain asks is (more or less) constant. The total cost is then the number of open channels (the edges of the network) times this constant cost to open each channel.

Our goal is to define a strategy for the PSP regarding which transactions to execute in order to maximize profit (fees from transactions minus costs to set up channels) and minimize capital (cryptomoney that is temporarily locked into channels). Note that there is a trade-off between profit and capital, as more capital may allow to accept more transactions, earning fees for each transaction, hence increasing profit. In particular, we discuss the following questions: What is the minimum capital needed to be able to accept a given set of transactions? What is the maximum profit we can achieve with a given capital? These questions are at the heart of understanding the Pareto-nature of the trade-off between profit and capital in payment networks.

1.2 Related Work

Current work on payment channels has mainly focused on designing routing algorithms for the implemented decentralized payment networks, such as the Lightning [16] and Raiden [2] networks. Prihodko et al. [17] present Flare, an efficient routing algorithm for the Lightning network by collecting information on the network’s local topology. Malavolta et al. [14] introduce the IOU credit network SilentWhispers where they use landmark routing to discover multiple paths and multi-party computation to decide the amount of capital to be locked on each path. Roos et al. [18] propose SpeedyMurmurs, a routing algorithm for payment networks that uses embedding-based path discovery to find routes from sender to receiver. However, all these protocols assume a network structure created by the individuals participating in the network. The goal is to discover the network topology and possible routes from sender to receiver of every transaction. Our objective is to design the optimal network structure assuming a central authority, the PSP.

An active line of research on payment channels is the construction of secure and private systems that can act as payment hubs. Heilman et al. [9] propose a Bitcoin-compatible construction of a payment hub for fast and anonymous off-chain transactions through
an untrusted intermediary. Green et al. [8] present Bolt (Blind Off-chain Lightweight Transactions) for constructing privacy-preserving unlinkable and fast payment channels. However, they do not analyze how expensive the construction of a payment hub is for a PSP. In this work, we answer the following questions: is a payment hub a good solution for a PSP? How much capital is required to build a payment hub compared to the capital of a capital-optimal network? These answers are highly relevant to the economic viability of a payment hub as a practical solution for payment networks, and ultimately whether payment networks can solve the eminent throughput problem of cryptocurrencies.

Our paper can be seen as a cryptocurrency variant of classic work on network design. It is as such somewhat related to fundamental work starting in the 1970s. For example, Johnson et al. [11] prove that given a weighted undirected graph, finding a subgraph that connects all the original vertices and minimizes the sum of the shortest path weights between all vertex pairs, subject to a budget constraint on the sum of its edge weights, is NP-hard. Another similar problem is the optimum communication spanning tree problem [10], whose input is a set of nodes, the distances and requests between them, and the goal is to find the spanning tree that minimizes the cost of communication (for each pair, the request multiplied by the sum of distance). Our channel design problem seems similar to these problems since the routing of a transaction matters, and our objective is to minimize the capital on the channels (like the original network design work wants to minimize the sum of the distances). However, in contrast with traditional network design, in payment networks the order of transactions matters, as the capital moves from one side of the channels to the other. Moving capital gives network design a surprising twist, as classic techniques do not work anymore. With the anticipated importance of payment networks, we believe one should have a fresh look at network design.

1.3 Our Contribution

We introduce an algorithmic framework for the channel network design problem. First, we study the offline problem, i.e., we are given the future sequence of transactions. We show that maximizing the profit given the capital assignments is NP-hard, even for a single channel. Then, we present a fully polynomial time approximation scheme for the single channel case. Later, we consider the case where the PSP wants to maximize its profit and thus execute all profitable transactions. We prove that a hub (a star graph) is a $2$-approximation with respect to the capital. Moreover, we show the problem is NP-complete under graph restrictions.

In addition, we examine online variants. First, we examine the online single channel case assuming the PSP wants to maximize its profit under capital constraints. We show that there is no deterministic competitive algorithm for adaptive adversaries. Later, we study the online channel design problem assuming all profitable transactions are executed. We show that the star graph yields an $O(\log C)$-competitive algorithm, where $C$ denotes the optimal capital.

2 Notation and Problem Variants

We assume the fee of a transaction on the blockchain to be constant, without loss of generality simply 1. The fee of a transaction in the payment network cannot be higher than the fee on the blockchain, or a potential user may prefer the blockchain over the payment network. A rational PSP will ask for a transaction processing fee which is as high as possible but lower than the blockchain fee, hence for $1 - \epsilon$. In our analysis we will usually assume that $\epsilon \to 0$.

Let us now formally define the problems we will study.
Problem 1 (General Payment Network Design).

Input: Capital $C$, profit $P$, the sequence of $n$ transactions $t_i = (s_i, r_i, v_i)$ with $1 \leq i \leq n$, each containing the sender node $s_i$, the receiver node $r_i$, and the value $v_i$ of the transaction $t_i$.

Output: Strategy $S = \{0, 1\}^n$, a binary vector where the $i^{th}$ position is 1 if we choose to execute the $i^{th}$ transaction of the input and 0 else. The graph $G(V, E, C_l, C_r)$ is the network we created to execute the chosen transactions, where $V$ is the set of senders and receivers that participate in any transaction, $E$ is the set of channels we open and $C_l, C_r$ the capital on each side of each edge. Each transaction can be routed arbitrarily in $G$, denoted by $S_e = \{-1, 0, 1\}^n$, for all $e \in E$, i.e., $S_e(i) = 1$ (or $-1$) if transaction $i$ is routed through edge $e$ from left to right (from right to left, respectively) and $S_e(i) = 0$ if transaction $i$ is not routed through edge $e$.

Our goal is to return (if it exists) a strategy $S$, a graph $G$ and a routing $S_e$ subject to the following constraints:

1. $|S| - |E| \geq P$
2. $\forall e \in E, \forall j \in \{1, 2, \ldots, n\}, -C_r(e) \leq \sum_{i=1}^{j} S_e(i) \cdot v_i \leq C_l(e)$
3. $\sum_{e \in E} C_l(e) + C_r(e) + |E| \leq C$

The first inequality guarantees that the fees of the accepted transactions minus the cost of opening the channels is at least as high as the intended profit. The second inequality makes sure that at any time the capital on each side of each channel is non-negative. The third inequality ensures that the used capital on the channels and the cost of opening the channels is at most the available capital.

Problem 1 in all its generality is difficult, as it features many variables. Consequentially, we mostly focus on the most interesting special cases of Problem 1: We consider transactions as possible. We focus on solving the problem for a single edge of the network, even a capital assignment) and we aim to maximize the PSP’s profit, hence execute as many transactions as possible. We focus on solving the problem for a single edge of the network, since even in this simple case the problem is challenging. Later, we focus on minimizing the capital given the PSP wants to execute all the profitable transactions.

3 Offline Channel Design

In this section, we study the offline channels network design problem, i.e., we assume we know the future transactions (for the next period). First, we explore the network topology for the general problem. Then, we examine the case where we are given a specific capital (or even a capital assignment) and we aim to maximize the PSP’s profit, hence execute as many transactions as possible. We focus on solving the problem for a single edge of the network, since even in this simple case the problem is challenging. Later, we focus on minimizing the capital given the PSP wants to execute all the profitable transactions.
3.1 Graph Topology

We first prove some observations concerning the optimal graph structure. We consider as optimal the solution that maximizes the profit while respecting the capital constrains (optimization version of Problem 1).

▶ Lemma 5. The graph of the optimal solution does not contain any node that sends and receives less than two transactions.

Thus, during preprocessing we can safely remove all transactions that contain a node that is only sender or receiver of a transaction in this one transaction. The time complexity of this procedure is linear in the number of transactions.

▶ Lemma 6. The optimal graph is not necessarily a tree (or forest).

Due to the complexity of the problem we focus on a single channel. It turns out that even for this degenerate case, the problem is far from trivial.

3.2 Single Channel

We now focus on a single channel. We prove that even in this case the problem of choosing the transactions that maximize the profit given capital assignments is NP-hard and present an FPTAS.

Specifically, we are given a sequence of transactions on a single edge of a network and their values, the capital assignment on the edge and a target profit. Our goal is to decide whether we can execute at least as many transactions as the given target profit while respecting the capital constraints. Since the number of edges is fixed and equal to 1 the profit now is the number of executed transactions (Problem 2). The problem is equivalent to a variant of the 0/1 knapsack problem where each transaction represents an item. Each item has profit 1 and either positive or negative size (values). The capacity of the knapsack is represented by the capital assignments and the goal is to maximize the profit while respecting the capacity.

▶ Problem 7 (Fixed Weight Subset Sum (FWSS)). Given a set of non-negative integers \( U = \{a_1, a_2, \ldots, a_n\} \), and non-negative integers \( A \) and \( l \), is there a non-empty subset \( U' \subseteq U \) such that \( |U'| = l \) and \( \sum_{i \in U'} a_i = A \)?

▶ Lemma 8. FWSS is NP-hard.

▶ Theorem 9. Problem 2 is NP-hard.

Proof. We will reduce Fixed Weight Subset Sum (FWSS) to Problem 2.

Assuming we are given an instance of the FWSS, we present a polynomial time transformation to an instance of Problem 2. We first define the capital assignment on the edge \( C_r(e) = A(l + 1) \), \( C_l(e) = 0 \) and the profit \( P = l + n(l + 1) \). Then, we define the sequence of transactions as follows: \( v_i = a_i + A, \forall 1 \leq i \leq n \) and \( v_i = -A/n, \forall n + 1 \leq i \leq n(l + 2) \). We will prove that there is a non-empty set that satisfies the FWSS problem if and only if we can choose transactions that satisfy the capital constraints and profit in the aforementioned instance.

Assume we have a “yes” instance of the problem. Then, we have chosen at least \( P = l + n(l + 1) \) transactions to execute. We will show that this corresponds to choosing \( l \) positive transactions that sum up to \( A(l + 1) \), thus to a solution of the FWSS problem. Towards contradiction, we examine the following three cases:
If the number of positive transactions is less than or equal to \( l \), the total profit is less than \( l + n(l + 1) \), since there are only \( n(l + 1) \) negative transactions.

If the number of positive transactions is more than \( l \), then we violate the capital constraints, since \( \sum v_i \geq A(l + 1) + \sum a_i > A(l + 1) = C_r(e) \), where \( i \) corresponds to the chosen transactions.

Suppose the \( l \) chosen transactions’ values sum to more than \( A(l + 1) \). Then, the capital constraint is violated.

Suppose the \( l \) chosen transactions’ values sum to less than \( A(l + 1) \); suppose the sum is \( Al + \sigma \) with some \( \sigma < A \). Then, then negative transactions to be executed can be at most \( \frac{A}{An} + \frac{\sigma}{An} < \ln n \). Thus, the profit is strictly less than \( l + \ln n + n \). Contradiction. Thus, a “yes” instance of our problem implies a “yes” instance of the FWSS problem. For the other direction, we will prove that if there is no subsequence of transactions of size at least \( P \) that satisfies the capital constraints, then there is no subset of size \( l \) that sums to \( A \) in FWSS. Equivalently, we will show that if there is a subset of size \( l \) that sums to \( A \) in FWSS, then there exists a subsequence of transactions of size at least \( P \) that satisfies the capital constraints. Suppose there is a non-empty set \( U' \subseteq U \) such that \( |U'| = l \) and \( \sum_{a_i \in U'} a_i = A \). Then we can execute the \( l \) transactions that correspond to the chosen \( a_i \)'s with exactly the \( C_r(e) \) capital, which will be transferred on \( C_l(e) = A(l + 1) \). Then, we can execute all the negative transactions since they are \( n(l + 1) \) many with values \( A/n \), thus we need \( A(l + 1) = C_l(e) \) capital. Therefore, we can execute \( P = l + n(l + 1) \) transactions, achieving the required profit while satisfying the capital constraints.

Both FWSS and Problem 2 are also polynomially verifiable, hence NP-complete.

The classic dynamic programming approach that typically yields a polynomial time algorithm when profits are fixed is not efficient since in this variation we cannot optimize using the minimum value at each step due to negative values. Instead, we present a fully polynomial time approximation scheme (FPTAS).

---

**Algorithm 1: MaxProfit.**

**Data:** number of transactions \( n \), values of the sequence of transactions \( v_i \in \mathbb{R}, \forall 1 \leq i \leq n \), capital \( C \), approximation factor \( \epsilon \).

**Result:** binary vector \( S = \{0, 1\}^n \) that indicates which transactions to execute.

Let \( K = \frac{C}{\epsilon n} \), where \( V = \max_{1 \leq i \leq n} v_i \); for all transactions \( 1 \leq i \leq n \) define \( v'_i = \lceil \frac{v_i}{\epsilon} \rceil \);

Let \( T(i,j) = 0 \), for all \( 1 \leq i \leq n \) and \( 1 \leq j \leq \frac{n^2}{\epsilon} \);

for \( i = 1 \) to \( n \) do

for \( j = 1 \) to \( \frac{n^2}{\epsilon} \) do

\[
T(i,j) = \begin{cases} 
\max\{T(i-1,j), T(i-1,j-v'_i)\}, & \text{if } \frac{C}{\epsilon} \geq j - v'_i > 0 \\
T(i-1,j), & \text{else}
\end{cases}
\]

Store for every \( T(i,j) \) a \( n \)-binary vector \( S_{i,j} \) that has value 1 in the \( k \)-th position if the \( k \)-th transactions is chosen to be executed;

end

Return vector \( S_{i,j} \) for the maximum \( T(i,j) \) such that \( \sum_{k=1}^{n} S_{i,j}(k) \cdot v_k \leq C \);
Theorem 10. Algorithm MaxProfit is a fully polynomial time approximation scheme for Problem 2.

Proof. The running time of the algorithm is $O\left(\frac{n^3}{\epsilon}\right)$, which is polynomial in both $n$ and $\frac{1}{\epsilon}$. We will prove that the profit of the output of algorithm MaxProfit is at least $(1 - \epsilon)$ times the optimal. We denote by $S$ the set of transactions returned by the algorithm, $O$ the set returning the optimal profit and $\text{prof}(X)$ the profit from the set of transactions $X$. Since we scaled down by $K$ and then rounded down, for every transaction $i$ we have that $Kv'_i \leq v_i$. Therefore, the optimal set’s profit can decrease at most $nK$, $\text{prof}(O) - \text{prof}'(O)K \leq nK$. The dynamic program returns the optimal set for the scaled instance. Thus, $\text{prof}(S) \geq \text{prof}'(O)K \geq \text{prof}(O) - nK = \text{prof}(O) - \epsilon V \geq (1 - \epsilon)\text{prof}(O)$, since $\text{prof}(O) \geq V$.

Scaling to many channels. Unfortunately, even when the graph is a tree, algorithm 1 does not scale efficiently. Creating an $m$-dimensional tensor for the dynamic program, where $m$ are the edges of the tree, has time complexity $O(C^m n)$ where $C$ is the maximum capital from all edges. Even if we bound the capital by a polynomial on $n$ the algorithm remains exponential due to the number of edges on the exponent. In the general case where the graph could contain cycles, the problem becomes even more complex. Now, we need to additionally consider all possible routes for each transaction; this adds an exponential factor on the running time of the algorithm.

Since Problem 1 is complex, we study special cases that might be useful in practice and provide an insight to the general problem.

3.3 Channel Design for Maximum Profit

In this section, our goal is to find the minimum capital for which we can achieve maximum profit, i.e., execute all profitable transactions (Problem 3). At first, we note some simple observations for the graph structure. Then, we prove that any star graph is a 2-approximate solution with respect to the capital, but even the “best” star is not an optimal solution. Last, we prove the problem is NP-hard when there are graph restrictions.

Throughout this section, we refer to the optimal solution of Problem 3 as the optimal network for maximum profit.

Lemma 11. When the capital is unlimited, the optimal network for maximum profit does not contain cycles.

Lemma 12. When the capital is unlimited, there exists an algorithm, with time complexity $\Theta(n)$, where $n$ denotes the number of transactions, that returns the optimal network for maximum profit.

Lemma 13. The optimal network for maximum profit is not necessarily a connected graph.

We refer to transactions that increase the PSP’s profit as profitable transactions. We assume all nodes participate in at least two transactions (Lemma 5).

Lemma 14. Not all transactions are profitable transactions.

Despite Lemma 13, we note that payment channels are monetary systems. As such, large companies are expected to participate in the network as highly connected nodes, ensuring that the optimal graph is one connected component. Thus, for the rest of the section we can safely assume that the optimal graph is connected.
We will now define some formal notation to prove that choosing any star as the graph to route all transactions requires at most twice the capital of the optimal graph. This immediately implies we have a 2-approximation to Problem 3.

Now, suppose we can update the capital of an edge before executing each transaction. This way we can guarantee there is enough capital on all channels for each transaction execution. These updates are for free, like assigning tokens, and we use them as a stepping stone to calculate the total capital (amortized analysis). Let us denote \( c_G(uv, i) \) the additional capital required at the edge \((u, v)\), for transaction \( t_i \) with direction from \( u \) to \( v \) on graph \( G \). Now, we have that the total capital on graph \( G \), denoted by \( C_G \), is

\[
C_G = \sum_{v(u,v)} \sum_{i} c_G(uv, t_i)
\]

Moreover, let \( opt \) denote the optimal graph and \( V \) the set of nodes involved in \( opt \). We will show that the capital used to route a sequence of transactions on any star that contains the same set of nodes as the optimal graph is at most twice the capital used by the optimal solution for the same sequence.

**Theorem 15.** Any star graph yields a 2-approximate solution for Problem 3.

**Proof.** To prove the theorem, we just need to prove that for any sequence of transactions \( t_1, t_2, \ldots, t_n \), for any star graph \( S(V) \), \( C_S \leq 2C_{opt} \). We will show that we can execute on the star graph the same sequence of transaction as the optimal solution with twice as many tokens (amortized capital). Initially we have zero tokens on all edges on both the optimal and the star graph. Every time a new transaction \( t_i \) comes the optimal solution finds a path from sender to receiver. For every edge \((u, v)\) on this path the optimal solution assigns \( c_{opt}(uv, t) \) tokens. Then, we assign on the star, \( S \), \( c_{opt}(uv, t) \) tokens on the edges \( mu \) and \( vm \), where \( m \) is the central node on \( S \). The only exceptions are the sender and receiver nodes, \( s \) and \( r \) respectively, where the tokens are initially placed on \( sm \) and \( mr \) to execute the transaction. Thus, for every transaction the sum of the tokens used on the star graph are twice the sum of the tokens used on the optimal solution. Therefore, the overall required capital on the star is at most twice the optimal capital, \( C_S \leq 2C_{opt} \).

To complete our proof, we need to show we assigned in total enough tokens to execute the given sequence of transactions. When a new transaction comes from \( s \) to \( t \), we only need to guarantee there enough tokens on \( sm \) and \( mt \). Obviously, if a transaction needs additional tokens to be executed on the optimal graph then the aforementioned strategy guarantees the additional tokens for the star graph as well. If there are already some tokens on the optimal graph for the sender then either he was previously an intermediate node or a receiver node. In both those cases the same amount of tokens would have been stored on \( sm \) as well. With a similar argument, if there were some tokens for the last edge to reach the receiver on the optimal graph then \( r \) was either an intermediate node or a sender. Again, in both those cases the same amount of tokens would have been assigned to \( mr \) on the star.

**Lemma 16.** The star graph is not an optimal solution for Problem 3.

**Discussion.** The centralized nature of the star is quite convenient for a payment network operated by a PSP. The star alleviates the problem of participation incentives detected on decentralized payment networks; now the participants of the network can be online only when they want to execute a transaction. Although the star graph is not optimal, it is a good enough solution for a PSP, since the capital he needs to lock in the channels is at most twice the minimum. Thus, payment hubs are an economically viable solution for the throughput problem on cryptocurrencies.
3.4 Channel Design with Graph Restrictions

An interesting variation of the problem is when the network has restrictions (Problem 4). Instead of allowing all possible channels, we assume some of them cannot occur in real life. In this case, we are given a graph with all the potential channels, the sequence of transactions and the capital, and we want to find the induced subgraph that maximizes the profit. We prove that the problem of deciding whether all given transactions can be executed in the given graph with a fixed capital is NP-complete.

The graph is given so the capital needed to open the channels is fixed in each given instance. Thus, we assume the capital corresponds solely to the capital we lock on the edges but not the one we require to open the channels.

Theorem 17. Problem 4 is NP-complete.

4 Online Channel Design

In this section, we study the online case, assuming no prior knowledge for the future transactions. When there is a transaction request we instantly decide whether to execute it or not through our network, assuming we have enough capital on the edges of the path we want to route the transaction. If there is not enough capital on some of the edges, we can refund a channel, which costs 1, the same as opening a new channel.

4.1 Single Channel with Capital Constraints

Similarly to the offline case, we first focus on the simpler case where we have a single edge and limited capital. The transactions arrive online, for each transaction we immediately decide whether it is accepted.

Theorem 18. There is no competitive algorithm for adaptive adversaries.

Proof. Suppose we have a channel with $C_r = C_l = 5$. Transactions from left to right have positive values, those from right to left have negative values. Let us consider two different transaction sequences:

1. $(1, 5, -10, 10, -10, 10, \ldots)$
2. $(1, 4, -10, 10, -10, 10, \ldots)$

Apart from the second transaction, both sequences are identical: The first transaction has value 1, starting with the third transaction we always move the complete capital with every transaction. The only difference is the second transaction.

If some online algorithm accepts the first transaction, then the adversary presents the first sequence; if the online algorithm denies the first transaction, then the adversary reveals the second sequence. Therefore, no matter whether this online algorithm accepts the first transaction or not, it can at most accept one transaction, while the optimal offline algorithm can accept almost all transactions (in case of the first sequence, the offline algorithm only needs to deny the first transaction, in case of the second sequence it will accept all transactions).

4.2 Channel Design for Maximum Profit

We assume again that we want to execute all transactions, thus the optimal graph does not contain cycles. Our objective is to minimize the capital, given all transactions will be executed through our payment network. Wlog, we assume the PSP is a node in the network.
Algorithm 2: OnlineMaxProfit.

Data: online sequence of transactions \( t_i = (s_i, r_i, v_i) \)

Result: capital \( C \)

We denote by \( s \) the node corresponding to the PSP.

\( E \leftarrow \emptyset \)
\( C \leftarrow 0 \)

for each transaction \( t_i \) do

if \( s_i \) is not connected to \( s \) then

\( E \leftarrow E \cup (s_i, s) \)
\( c_{s_i, s} \leftarrow v_i \)
\( C \leftarrow C + 1 \)

else if \( c_{s_i, s} < v_i \) then

\( c_{s_i, s} \leftarrow c_{s_i, s} + v_i \)
\( c_{s, s_i} \leftarrow c_{s, s_i} + v_i \)
\( C \leftarrow C + 1 \)

else

\( c_{s_i, s} \leftarrow c_{s_i, s} - v_i \)
\( c_{s, s_i} \leftarrow c_{s, s_i} + v_i \)

end

For the case of \( r_i \) we follow a similar (invert) procedure.

end

for all \( i \neq s \) do

\( C \leftarrow C + c_{i, s} + c_{s, i} \)

end

Return capital \( C \)

Similarly to the offline case, we show that constructing a star network to connect the nodes with payment channels is a good solution. Specifically, we present a log-competitive algorithm that takes advantage of the star graph structure.

In Algorithm OnlineMaxProfit, we gradually form a star where the center is the PSP. At each step, we check whether there is enough capital on the edges to and from the center to execute the current transaction. If the capital on an edge is smaller that the value of the current transaction, we refund the channel and add to the capacity of this edge twice the value of the current transaction.

▶ Theorem 19. Algorithm OnlineMaxProfit is \( \Theta(\log C_{opt}) \)-competitive.

Proof. The star is a 2-approximation to the optimal offline solution, thus we start with a competitive ratio of 2. The way we update the capacities, each time adding twice the value of the transaction if the capacity is less than the transaction’s value, yields also a competitive ratio of two on the edges’ capacities. Moreover, at each such step we at least double the capacity of an edge thus we reach the edge’s optimal capital, \( C_e \), in \( \log C_e \) steps. If we sum over all edges, in total we refund the channels at most \((n - 1) \log C_{edges}\) times, where \( n \) is the number of nodes in the network and \( C_{edges} \) the edges’ optimal capital of the offline solution. Therefore, algorithm OnlineMaxProfit returns \( C \leq (n - 1) \log C_{edges} + 4C_{edges} \), while the offline solution requires \( C_{opt} = (n - 1) + C_{edges} \). This yields a competitive ratio of \( \Theta(\log C_{opt}) \).
5 Conclusion

We introduced a graph theoretic framework for payment networks. We studied the problem for a specific epoch, i.e., for a fixed number of transactions. This restriction is due to privacy issues, such as timing attacks on the payment network that can leak information on the customers’ personal data. We tried to maximize the profit (the number of accepted transactions minus the number of generated channels) and to minimize the capital needed to execute these transactions. Due to the multi-objective nature, there are several versions of this problem. In this paper, we mainly focused on two interesting variations:

1. How to choose transactions to execute on a single channel with given capital assignments to maximize the profit,
2. How to design a network and assign capitals to accept all transactions and minimize the needed capital.

It turns out, these two problems are challenging, as we show that the first problem and a variation of the second one are both NP-hard. We propose a dynamic programming based algorithm for the single channel problem and show that it is an FPTAS. For the network design and capital assignment problem, we show that stars achieve approximation ratio $\frac{2}{3}$. In other words, hubs are not only an implementable and privacy-guaranteed solution, as mentioned in [9] and [8], but also a satisfactory solution for PSP from the profit-maximization point of view.

We also studied the online versions of these problems. For the single channel case we show that it is impossible to design a competitive algorithm against an adaptive adversary. For the online channel design for maximum profit, we devise an $O(\log C)$-competitive online algorithm based on the star structure.

The results presented in this paper and the proposed algorithms can be applied to other fields such as traffic network design. For example, every airline would want to maximize the profit and to minimize the costs (of creating new routes and purchasing new airplanes). Interestingly, similar to what we discovered, hubs are indeed used by almost all airlines, e.g., most flights of the Turkish airline departure from or fly to Istanbul.

Apart from capital assignment, fee assignment of payment networks [3] is also related to the traffic network design problem. One need to pay for using highways in some countries (e.g., Greece, China and France), thus the companies need to decide which cities are connected by highways and how much one needs to pay for every path. In this way, the drivers prefer highways (analog to the payment channels) to other slow paths (analog to the main chain), and hence the profit is maximized.

References

Algorithmic Channel Design


Counting Connected Subgraphs with Maximum-Degree-Aware Sieving

Andreas Björklund
Department of Computer Science, Lund University, Sweden

Thore Husfeldt
BARC, IT University of Copenhagen, Denmark and Lund University, Sweden

Petteri Kaski
Department of Computer Science, Aalto University, Finland

Mikko Koivisto
Department of Computer Science, University of Helsinki, Finland

Abstract

We study the problem of counting the isomorphic occurrences of a $k$-vertex pattern graph $P$ as a subgraph in an $n$-vertex host graph $G$. Our specific interest is on algorithms for subgraph counting that are sensitive to the maximum degree $\Delta$ of the host graph.

Assuming that the pattern graph $P$ is connected and admits a vertex balancer of size $b$, we present an algorithm that counts the occurrences of $P$ in $G$ in $O\left((2\Delta - 2)^{\frac{1}{2}\frac{k}{2}} 2^{-b} \frac{n}{2} k^2 \log n\right)$ time.

We define a balancer as a vertex separator of $P$ that can be represented as an intersection of two equal-size vertex subsets, the union of which is the vertex set of $P$, and both of which induce connected subgraphs of $P$.

A corollary of our main result is that we can count the number of $k$-vertex paths in an $n$-vertex graph in $O\left((2\Delta - 2)^{\frac{1}{2}\frac{k}{2}} 2^{-b} \frac{n}{2} k^2 \log n\right)$ time, which for all moderately dense graphs with $\Delta \leq n^{1/3}$ improves on the recent breakthrough work of Curticapean, Dell, and Marx [STOC 2017], who show how to count the isomorphic occurrences of a $q$-edge pattern graph as a subgraph in an $n$-vertex host graph in time $O(q^{q0.17q})$ for all large enough $q$. Another recent result of Brand, Dell, and Husfeldt [STOC 2018] shows that $k$-vertex paths in a bounded-degree graph can be approximately counted in $O(4^k n)$ time. Our result shows that the exact count can be recovered at least as fast for $\Delta < 10$.

Our algorithm is based on the principle of inclusion and exclusion, and can be viewed as a sparsity-sensitive version of the “counting in halves”-approach explored by Björklund, Husfeldt, Kaski, and Koivisto [ESA 2009].

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases graph embedding, $k$-path, subgraph counting, maximum degree

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.17

Funding This work was supported in part by the Swedish Research Council grant VR-2016-03855, “Algebraic Graph Algorithms”. BARC, Basic Algorithms Research Copenhagen, is funded by the Villum Foundation grant 16582. The research leading to these results has received funding from the European Research Council under the European Union’s Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement 338077 “Theory and Practice of Advanced Search and Enumeration”.

1 Introduction

Subgraph statistics are among the most fundamental and extensively studied invariants of graphs. A canonical task in this domain is to count the number of isomorphic occurrences of a connected $k$-vertex pattern graph as a subgraph in an $n$-vertex host graph.

Assuming $k$ is a constant, we can explicitly list all the occurrences of the pattern in the host graph in time $O(n^k)$. A substantial literature exists on counting algorithms that improve on the $O(n^k)$ bound. Currently the fastest algorithm design for the general case of an unconstrained $k$-vertex pattern remains the $O(n^{\omega k/3})$-time algorithm of Nešetřil and Poljak [27] (see also Eisenbrand and Grandoni [14]), where $2 \leq \omega < 2.3728639$ is the exponent of $n \times n$ matrix multiplication (cf. Le Gall [24] and Vassilevska Williams [30]). By parameterizing on the structure of the pattern graph, many further and faster algorithm designs become possible; we postpone a detailed discussion of earlier work after a statement of our present focus and main result.

Sensitivity to the maximum degree. In this paper, we are interested in subgraph-counting algorithms that are sensitive to the maximum degree $\Delta$ in addition to the number of vertices $n$ in the host graph. Our interest is in particular on algorithm designs that scale to massive graphs where $\Delta$ can be orders of magnitude smaller than $n$. Such study of algorithms that are sensitive to $\Delta$ can be found, for example, in the work of Komusiewicz and Soren [23] in the context of optimization over all $k$-vertex subgraphs.

In our case of connected subgraph counting, it is immediate that the host can contain at most $n(\Delta - 1)^k$ subgraphs that are isomorphic to the pattern, and furthermore these subgraphs can be trivially listed in $O(n(\Delta - 1)^k)$ time.

Our goal in this paper is to improve the trivial running time of connected subgraph counting to the general form

$$O(n(\alpha \Delta)^{\beta k})$$

for constants $\alpha \geq 1$ and $0 \leq \beta \leq 1$ that depend on the topology of the pattern but not on the parameters $k$, $n$, and $\Delta$. In particular, the main conceptual contribution of this paper is to establish that nontrivial $\Delta$-sensitive exponents $\beta < 1$ can be achieved for elementary connected topologies, such $k$-vertex paths and cycles, for which we establish $\beta = 1/2$ and $\alpha = 2$ independently of $k$ (cf. Corollary 3 for a precise statement). Furthermore, our algorithms scale linearly in the number of vertices $n$, thus enabling a more fine-grained control of subgraph counting by isolating the complexity to the maximum degree $\Delta$ and the topology of the connected pattern.

Our results. Let us now proceed with a detailed statement of our results. (The standard graph-theoretic terminology and preliminaries can be found in Section 2.) We are interested in connected pattern graphs that admit a small balancer in the following sense.

Definition 1 (Balancer). A vertex subset $B \subseteq V(P)$ of a connected graph $P$ is a balancer if there exist subsets $C_1, C_2 \subseteq V(P)$ such that

1. $|C_1| = |C_2|$,  
2. $C_1 \cup C_2 = V(P)$,  
3. the induced subgraphs $P|C_1$ and $P|C_2$ are connected, and  
4. $B = C_1 \cap C_2$ is a $(C_1 \setminus C_2, C_2 \setminus C_1)$-separator in $P$.

1 To avoid degenerate cases, let us assume $\Delta \geq 2$ in what follows.
For example, a \(k\)-vertex path has a balancer of size \(2 - (k \mod 2)\), a \(k\)-vertex cycle has a balancer of size \(2 + (k \mod 2)\), and a \(k\)-vertex tree for \(k \geq 3\) has a balancer of size at most \㎞/₃\) (cf. Lemma 7). Trivially, every \(k\)-vertex connected graph has a balancer of size at most \(k\). It is also immediate that a balancer and \(k\) must have the same parity.

Theorem 2 (Main; Counting connected subgraphs with a small balancer). Let \(P\) be a connected \(k\)-vertex graph with a balancer of size \(b\), and let \(G\) be an \(n\)-vertex graph with maximum degree \(\Delta \geq 2\). There exists an algorithm that counts the number of isomorphic occurrences of \(P\) as a subgraph in \(G\) in time

\[
O\left((2\Delta - 2)\frac{4\pi}{\pi} 2^{-b} n^{k/2} \log n\right).
\]

(2)

Let us illustrate the use of Theorem 2 by stating a corollary for elementary connected patterns such as paths, cycles, and trees with arbitrary topology.

Corollary 3. There exist algorithms that output, given as input an \(n\)-vertex host graph \(G\) of maximum degree \(\Delta \geq 2\),

1. the number of \(k\)-vertex paths in \(G\) in time \(O((2\Delta - 2)\frac{4\pi}{\pi}nk^2 \log n)\),
2. the number of \(k\)-vertex cycles in \(G\) in time \(O((2\Delta - 2)\frac{4\pi}{\pi}nk^2 \log n)\), and
3. the number of isomorphic occurrences of any fixed \(k\)-vertex tree \(T\) for \(k \geq 3\) in \(G\) in time \(O((2\Delta - 2)\frac{4\pi}{\pi}nk^2 \log n)\).

Discussion and related work. Recently, Patel and Regts [29] have shown that the number of subgraphs of \(G\) that induce an isomorphic occurrence of a given \(k\)-vertex pattern graph can be computed in time \(\Delta^{O(k)}n\); their precise bound is \(O((n(4\Delta)^{2k} + 2^{10k})\text{poly}(k))\). This result is sensitive to the sparsity of the host graph even when the pattern graph is disconnected.

Just as recently, Curticapean, Dell, and Marx [12] showed that the isomorphic occurrences of a \(q\)-edge pattern graph in an \(n\)-vertex host graph can be counted for all large enough \(q\) in \(O(q^2n^{0.17q})\) time, building on the connection between the number of subgraphs and the number of homomorphisms established by Lovász 50 years ago [25]. As further motivation for our present study, we observe that the Curticapean–Dell–Marx algorithm cannot in a direct way utilise sparsity of the host graph, even when the pattern graph is connected. Indeed, the Curticapean–Dell–Marx algorithm is based on homomorphism-counting over low-width tree-decompositions of consolidations of the pattern graph, and there is no guarantee that the bags of such a tree-decomposition induce connected subgraphs, which means their algorithm has to track essentially arbitrary mappings of vertices to the host graph. In contrast, our present algorithm tracks embeddings of connected subgraphs of the pattern graph, which enables us to control the number of such embeddings with \(\Delta\). For \(k\)-vertex paths, or any connected pattern graph with a balancer much smaller than \(k\), we obtain a faster subgraph counting algorithm for every \(\Delta \leq n^{1/3}\).

In another very recent work, Brand, Dell, and Husfeldt [8] show that one can approximately count the number of \(k\)-vertex paths in a bounded-degree \(n\)-vertex host graph in \(O(4^n)\) time with a randomized approximation scheme. That is, for every \(\epsilon > 0\) they present a Monte-Carlo algorithm that computes, with probability at least \(\frac{99}{100}\), a factor-(\(1 + \epsilon\)) approximation of the exact count, with running time inversely proportional to \(\epsilon^2\). Our algorithm is deterministic and recovers the exact count in the same time, or faster, for all graphs with \(\Delta < 10\).

A third improvement concerns deterministic \(k\)-path detection. Zehavi [33] shows that there is a deterministic algorithm for \(k\)-path detection in general graphs running in \(O(2.6^k \text{poly}(n))\) time. No better algorithm is known for \(\Delta = 4\). For \(\Delta = 3\), one can enumerate the non-backtracking walks in \(O(2^k n)\) time. For \(\Delta = 4\), we directly obtain a \(O(2.44^k n)\) time algorithm as a special case of Corollary 3.
Methodology. Our algorithmic insight here is an old one: that one can use a meet-in-the-middle approach dividing the pattern graph (or more precisely in our present case, an embedding of the pattern graph) into two equal halves. To count half-pairs that together define an embedding of the pattern into the host, we can use an inclusion–exclusion sieve that cancels every pair with overlaps outside a controlled root. Björklund et al. [6] showed one can count the occurrences of a subgraph in $n^{k/2}$ time using fast zeta transforms and an inclusion–exclusion sieve on the subset lattice. However, as far as we can tell, one cannot exploit sparsity in this sieve directly. Our new algorithm here is based on observing that many of the computation points on the sieve will be zero. Rather than applying a fast zeta transform, we are better off by explicitly computing the points where the result is non-zero.

Further earlier work and complexity results. Subgraph counting has received a substantial amount of attention in the algorithms community. A non-exhaustive sample of earlier work includes Itai and Rodeh [19], Nešetřil and Poljak [27], Alon, Yuster, Zwick [3], Alon and Gutner [2], Eisenbrand and Grandoni [14], Björklund, Husfeldt, Kaski, and Koivisto [6], Björklund, Kaski, and Kowalik [7], Vassilevska Williams, Wang, Williams, and Yu [31], Vassilevska Williams and Williams [32], Amini, Fomin, and Saurabh [4], Fomin, Lokhtanov, Raman, Saurabh, and Raghavendra Rao [17], Floderus, Kowaluk, Lingas, and Lundell [15], Olariu [28], Kloks, Kratsch, Müller [22], Curticapean, Dell, and Marx [12], Brand, Dell, and Husfeldt [8], and Austrin, Kaski, and Kubjas [5].

From a parameterized complexity perspective, subgraph counting parameterized by the number of vertices $k$ in the pattern graph $P$ is a hard problem in the class #W[1] when $P$ has unbounded vertex cover number. Cf. Flum and Grohe [16], Chen and Flum [9], Chen, Thurley, Weyer [10], Curticapean [11], Curticapean and Marx [13], Jerrum and Meeks [20,21], and Meeks [26]. The specific problem of finding and counting cliques is used as a source of fine-grained hardness reductions by Abboud, Backurs, and Vassilevska Williams [1].

Under the Exponential Time Hypothesis (ETH), Impagliazzo et al. [18] have shown that there can be no algorithm for detecting a Hamiltonian path in time $\exp o(n)$. By inspecting the textbook reduction from 3-Satisfiability to Hamiltonicity used in that argument, we observe that this result holds even if the input graph has constant degree. Thus, the constant $\beta$ in (1) cannot be arbitrarily reduced, even if the dependency on $\Delta$ is much relaxed. In particular, the hypothesis forbids an algorithm for counting (or even detecting) $k$-paths with running time $(f(\Delta))^{\alpha(k)}\text{poly}(n)$ for any computable function $f$.

Organization. The rest of this paper is organized as follows. Section 2 reviews the standard definitions and notational conventions used in this paper. Section 3 presents our main sieving lemma for counting embeddings from two parts. Section 4 gives an algorithm for listing the embeddings of a connected pattern graph to a host graph. Section 5 develops our sieving algorithm for counting embeddings from two parts. Section 6 completes our main algorithm design and the proof of Theorem 2. Section 7 proves Corollary 3 and studies balancers in elementary families of connected graphs.

2 Preliminaries

This section reviews the standard definitions and notational conventions used in this paper.

Graphs and subgraphs. Unless mentioned otherwise, all graphs in this paper are undirected, loopless, and without parallel edges. For a graph $G$, we write $V = V(G)$ for the vertex set and $E = E(G)$ for the edge set of $G$, where each edge $e \in E(G)$ is a 2-element subset of
V(G). Let us write $\Delta = \Delta(G)$ for the maximum degree of a vertex in $G$. A graph $H$ is a subgraph of a graph $G$ if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. We write $H \subseteq G$ to indicate that $H$ is a subgraph of $G$. For a set $S \subseteq V(G)$, the subgraph $G[S]$ of $G$ induced by $S$ is defined by $V(G[S]) = S$ and $E(G[S]) = \{\{u, v\} \in E(G) : u, v \in S\}$. We tacitly assume in what follows that all algorithms accept their input graphs in adjacency list form.

Separators. Let $G$ be a graph and let $A, B \subseteq V(G)$. We say that a set $S \subseteq V(G)$ is an $(A, B)$-separator in $G$ if for all $a \in A$ and all $b \in B$ it holds that every path in $G$ joining $a$ and $b$ contains at least one vertex in $S$.

Mappings. For a mapping $\varphi : X \rightarrow Y$ and a subset $S \subseteq X$, we write $\varphi|_S : S \rightarrow Y$ for the restriction of $\varphi$ to $S$ and $\varphi(S) = \{\varphi(x) : x \in S\} \subseteq Y$ for the image of $S$ under $\varphi$. For two mappings $\varphi : X \rightarrow Y$ and $\psi : Y \rightarrow Z$, let us write $\psi \circ \varphi : X \rightarrow Z$ for their composition defined for all $x \in X$ by $\psi \circ \varphi(x) = \psi(\varphi(x))$.

Homomorphism, embedding, isomorphism, automorphism. Let $P$ and $G$ be graphs. A mapping $\varphi : V(P) \rightarrow V(G)$ is a homomorphism from $P$ to $G$ if for all $\{u, v\} \in E(P)$ it holds that $\{\varphi(u), \varphi(v)\} \in E(G)$. An injective homomorphism is called an embedding (or a monomorphism) of $P$ into $G$. A bijective homomorphism whose inverse is also a homomorphism is an isomorphism. An isomorphism from a graph $P$ to itself is an automorphism of $P$.

Let us write $\text{Hom}(P, G)$, $\text{Emb}(P, G)$, $\text{Iso}(P, G)$ for the set of all homomorphisms, embeddings, and isomorphisms, respectively, from $P$ to $G$. Similarly, let us write $\text{Aut}(P)$ for the set of all automorphisms of $P$.

Subgraph occurrences and subgraph counting. Let $P$ and $G$ be graphs. Let us write $\text{Sub}(P, G)$ for the set of all subgraphs $H \subseteq G$ such that $P$ and $H$ are isomorphic. We call each element of $\text{Sub}(P, G)$ an occurrence of $P$ in $G$. The number of embeddings of $P$ to $G$ and the number of occurrences of $P$ in $G$ are related by the identity

$$|\text{Emb}(P, G)| = |\text{Aut}(P)| \cdot |\text{Sub}(P, G)|. \quad (3)$$

In particular, assuming $|\text{Aut}(P)|$ is known, knowledge of one of $|\text{Emb}(P, G)|$ or $|\text{Sub}(P, G)|$ enables one to solve for the other via (3).

Iverson bracket notation. For a logical proposition $P$, it will be convenient to use Iverson’s bracket notation

$$[P] = \begin{cases} 1 & \text{if } P \text{ is true;} \\ 0 & \text{if } P \text{ is false.} \end{cases}$$

Model of computation. We work in a word-RAM model where basic word operations on $O(\log n)$-bit words take time $O(1)$, where $n = |V(G)|$ is the number of vertices in the input host graph $G$.

3 A sieving lemma for the number of embeddings

This section starts our work towards the proof of Theorem 2. The goal of this section is a technical sieving lemma that enables us to count embeddings $\varphi$ “in halves” (in analogy with Björklund et al. [6]) by sieving pairs $(\varphi_1, \varphi_2)$ of partial embeddings for those pairs that both agree with a root map $\rho$ and are otherwise disjoint in terms of their image sets.
In more precise terms, let \( P \) and \( G \) be graphs and let \( C_1, C_2 \subseteq V(P) \) such that
1. \( C_1 \cup C_2 = V(P) \) and
2. \( C_1 \cap C_2 \) is a \((C_1 \setminus C_2, C_2 \setminus C_1)\)-separator in \( P \).

Let us fix a root map \( \rho : C_1 \cap C_2 \to V(G) \). We say that an embedding \( \varphi \in \text{Emb}(P,G) \) is \( \rho \)-rooted if \( \varphi|_{C_1 \cap C_2} = \rho \). Let us write \( \text{Emb}_\rho(P,G) \) for the set of all \( \rho \)-rooted embeddings in \( \text{Emb}(P,G) \).

The following sets will form the core of the sieve. For \( X \subseteq V(G) \) and \( S \subseteq V(P) \) with \( C_1 \cap C_2 \subseteq S \), let us define

\[
I_{\rho,S}(X) = \{ \varphi \in \text{Emb}_\rho(P[S],G) : X \subseteq \varphi(S) \}. 
\]

We are now ready for our main sieving lemma.

**Lemma 4** (Sieving \( \rho \)-rooted embeddings from two parts). We have

\[
|\text{Emb}_\rho(P,G)| = \sum_{X \subseteq V(G)|\rho(C_1 \cap C_2)} (-1)^{|X|} \cdot |I_{\rho,C_1}(X)| \cdot |I_{\rho,C_2}(X)|. 
\]

**Proof.** Recalling that every nonempty finite set has equally many even-sized and odd-sized subsets, for any (possibly empty) finite set \( Y \) we have

\[
\sum_{X \subseteq Y} (-1)^{|X|} = \|Y = \emptyset\|. 
\]

Let us use the notational shorthands \( V_\rho = V(G) \setminus \rho(C_1 \cap C_2) \), \( E_1 = \text{Emb}_\rho(P[C_1],G) \), and \( E_2 = \text{Emb}_\rho(P[C_2],G) \). Expanding the right-hand side of (5), we obtain

\[
\sum_{X \subseteq V_\rho} (-1)^{|X|} \cdot |I_{\rho,C_1}(X)| \cdot |I_{\rho,C_2}(X)| \\
= \sum_{X \subseteq V_\rho} (-1)^{|X|} \sum_{\varphi_1 \in E_1} \|X \subseteq \varphi_1(C_1)\| \sum_{\varphi_2 \in E_2} \|X \subseteq \varphi_2(C_2)\| \\
= \sum_{\varphi_1 \in E_1} \sum_{\varphi_2 \in E_2} \sum_{X \subseteq V_\rho} (-1)^{|X|}[X \subseteq \varphi_1(C_1)][X \subseteq \varphi_2(C_2)] \\
= \sum_{\varphi_1 \in E_1} \sum_{\varphi_2 \in E_2} \sum_{X \subseteq V_\rho} (-1)^{|X|}[X \subseteq \varphi_1(C_1) \cap \varphi_2(C_2)] \\
= \sum_{\varphi_1 \in E_1} \sum_{\varphi_2 \in E_2} \|\varphi_1(C_1 \setminus C_2) \cap \varphi_2(C_2 \setminus C_1) = \emptyset\|.
\]

To establish the lemma, it now suffices to show that the last double sum equals \(|\text{Emb}_\rho(P,G)|\).

Toward this end, let us observe that a pair \((\varphi_1, \varphi_2) \in E_1 \times E_2\) of embeddings defines a unique embedding \( \varphi \in \text{Emb}_\rho(P,G) \) if and only if we have

\[
\varphi_1(C_1 \setminus C_2) \cap \varphi_2(C_2 \setminus C_1) = \emptyset. 
\]

In the “if” direction, each pair \((\varphi_1, \varphi_2) \in E_1 \times E_2\) defines a map \( \varphi : V(P) \to V(G) \) via the restrictions \( \varphi|_{C_1} = \varphi_1 \) and \( \varphi|_{C_2} = \varphi_2 \). Indeed, we observe that \( \varphi \) is a well-defined injective map because we have \( \varphi_1|_{C_1 \cap C_2} = \varphi_2|_{C_1 \cap C_2} = \rho \) together with \( C_1 \cup C_2 = V(P) \) and (7).
Furthermore, $\varphi$ is a homomorphism from $P$ to $G$ because $\varphi_1, \varphi_2$ are homomorphisms and because $C_1 \cap C_2$ is an $(C_1 \setminus C_2, C_2 \setminus C_1)$-separator in $P$; that is, $\varphi$ maps every edge of $P$ to an edge of $G$ since every edge of $P$ has both of its end-vertices in $C_1$ or in $C_2$.

In the “only if” direction, each embedding $\varphi \in \text{Emb}_q(P, G)$ restricts to $\varphi_1 = \varphi|_{C_1}$ and $\varphi_2 = \varphi|_{C_2}$. It is immediate that we have $\varphi_1 \in E_1$, $\varphi_2 \in E_2$, and (7) holds. This completes the lemma. ▶

**Remark.** From (4) it is immediate that we have $I_{P,C_j}(X) = \emptyset$ unless $|X| \leq |C_j|$, so it suffices to restrict the sieve (5) to sets $X$ with $|X| \leq \min(|C_1|, |C_2|)$.

## 4 Listing the embeddings of a connected pattern graph

To turn Lemma 4 into an algorithm that is sensitive to the maximum degree $\Delta = \Delta(G)$ of the host graph $G$, we will rely on a subroutine that we use to explicitly list the embeddings in $\text{Emb}(P[C_1], G)$ and in $\text{Emb}(P[C_2], G)$. This $\Delta$-sensitive listing subroutine is the content of this section and the following lemma.

**Lemma 5** (Listing embeddings of a connected pattern graph). Let $Q$ be a connected graph with $q = |V(Q)|$. Let $G$ a graph with $n = |V(G)|$ and $\Delta = \Delta(G) \geq 2$. There exists an algorithm that lists all the embeddings in $\text{Emb}(Q, G)$ in time

$$O\left(n(\Delta - 1)^{q-1}q^2\log n\right). \tag{8}$$

**Proof.** Since $Q$ is connected, it has a spanning tree. Fix an arbitrary spanning tree $T$ of $Q$ and fix an arbitrary vertex $r \in V(T)$ as the root of $T$. Use a recursive procedure to construct all embeddings $\varphi : V(T) \to V(G)$ one image $\varphi(x) \in V(G)$ at a time for each $x \in V(T)$, starting from the root $r$, and proceeding so that whenever the image of $x \neq r$ is being fixed, the parent $p \in V(T)$ of $x$ in $T$ (towards the root $r$) has its image $\varphi(p)$ already fixed. Whenever an embedding $\varphi : V(T) \to V(G)$ is completed, we test whether $\varphi$ is an embedding of $Q$ to $G$ and output $\varphi$ if this is the case.

To analyze the running time, we observe that there are at most $n$ choices for the image $\varphi(r) \in V(G)$ of the root $r$. Since the next image needs to be adjacent to $\varphi(r)$, there are at most $\Delta$ choices for the next image (if any). For all subsequent $q - 2$ images (if any), we have that there are at most $\Delta - 1$ choices for $\varphi(x)$ since $\varphi(x)$ and $\varphi(p)$ are adjacent and $\varphi(x)$ needs to be distinct from all the previously fixed images. Thus, in total there are at most $n\Delta(\Delta - 1)^{q-2} = O(n(\Delta - 1)^{q-1})$ embeddings $\varphi \in \text{Emb}(T, G)$. The (unoptimized) component $q^2\log n$ in the running time bound (8) comes from testing that the $|E(Q)| \leq q^2$ adjacencies $\{\varphi(z), \varphi(w)\} \in E(G)$ hold for each $\{z, w\} \in E(Q)$ by binary search to the adjacency lists of $G$ given by $\varphi$. ▶

**Remark.** We observe that the listing algorithm in Lemma 5 would not work without the assumption that $Q$ is connected.

## 5 A sieving algorithm for the number of embeddings

This section continues our work towards Theorem 2 by combining Lemma 4 and Lemma 5 to a sieving algorithm for the number $|\text{Emb}(P, G)|$ of embeddings of a connected $k$-vertex pattern graph $P$ to an $n$-vertex host graph $G$.

The sieving algorithm will rely on a balancer for $P$. In more precise terms, let $C_1, C_2 \subseteq V(P)$ so that $B = C_1 \cap C_2$ is a balancer of size $b = |B|$. Recalling Definition 1, we have
1. \(|C_1| = |C_2|\).
2. \(C_1 \cup C_2 = V(P)\).
3. the induced subgraphs \(P[C_1]\) and \(P[C_2]\) are connected, and
4. \(C_1 \cap C_2\) is a \((C_2 \setminus C_1)\)-separator in \(P\).

Furthermore, since \(k = |V(P)|\) and \(b = |C_1 \cap C_2|\), we thus have \(|C_1| = |C_2| = \frac{k+b}{2}\).

\textbf{Lemma 6} (Sieving algorithm for the number of embeddings). Let \(P\) be a connected \(k\)-vertex graph with a balancer of size \(b\). Let \(G\) be a graph with \(n = |V(G)|\) and \(\Delta = \Delta(G) \geq 2\). There exists an algorithm that computes \(|\text{Emb}(P,G)|\) in time

\[O\left((2\Delta - 2)^{\frac{\Delta+b}{2}} 2^{-b} \frac{n}{\Delta} k^2 \log n\right).\] (9)

\textbf{Proof.} Let \(C_1, C_2\) be the sets in that define the balancer of size \(b\). Recall the sets (4) that form the core of the sieve in Lemma 4. The algorithm works with a dictionary data structure that records and builds the nonempty sets \(I_{\rho,C_j}(X)\) indexed by three-tuples \((\rho, C_j, X)\) with \(\rho : C_1 \cap C_2 \to V(G)\), \(j = 1, 2\), and \(X \subseteq V(G) \setminus \rho(C_1 \cap C_2)\). We build the nonempty sets \(I_{\rho,C_j}(X)\) using the listing algorithm in Lemma 5.

First, we use the algorithm in Lemma 5 with \(Q = P[C_1]\) and \(|V(Q)| = \frac{k+b}{2}\) to list all the embeddings \(\varphi_1 \in \text{Emb}(P[C_1], G)\). By the analysis in Lemma 5, there are at most \(n(\Delta - 1)^{\frac{\Delta+b}{2}}\) such embeddings. For each listed \(\varphi_1\), we insert \(\varphi_1\) into the set \(I_{\rho,C_1}(X)\) for \(\rho = \varphi_1|_{C_1 \cap C_2}\) and for each \(X \subseteq \varphi_1(C_1 \setminus C_2)\). Since \(|\varphi_1(C_1 \setminus C_2)| = |C_1 \setminus C_2| = \frac{k-b}{2}\), the number of nonempty sets \(I_{\rho,C_1}(X)\) will be at most

\[n(\Delta - 1)^{\frac{\Delta+b}{2}} 2^{-b} = O\left(\frac{n}{\Delta}(2\Delta - 2)^{\frac{\Delta+b}{2}} 2^{-b}\right).\] (10)

Second, we use the algorithm in Lemma 5 with \(Q = P[C_2]\) and \(|V(Q)| = \frac{k+b}{2}\) to list all the embeddings \(\varphi_2 \in \text{Emb}(P[C_2], G)\). For each listed \(\varphi_2\), we insert \(\varphi_2\) into the set \(I_{\rho,C_2}(X)\) for \(\rho = \varphi_2|_{C_1 \cap C_2}\) and for each \(X \subseteq \varphi_2(C_2 \setminus C_1)\). The number of nonempty sets \(I_{\rho,C_2}(X)\) will similarly be at most (10).

Third, let us observe that we have used total time (10) and have available all nonempty sets \(I_{\rho,C_1}(X)\) and \(I_{\rho,C_2}(X)\). From Lemma 4 we observe that

\[|\text{Emb}(P,G)| = \sum_{\rho : C_1 \cap C_2 \to V(G)} |\text{Emb}_\rho(P,G)| = \sum_{\rho : C_1 \cap C_2 \to V(G)} \sum_{X \subseteq V(G) \setminus \rho(C_1 \cap C_2)} (-1)^{|X|} \cdot |I_{\rho,C_1}(X)| \cdot |I_{\rho,C_2}(X)|.\] (11)

Thus, we can compute \(|\text{Emb}(P,G)|\) using the nonempty sets \(I_{\rho,C_1}(X)\) and \(I_{\rho,C_2}(X)\) by sorting the index tuples based first on \(\rho\) and then based on \(X\). We then evaluate \(|\text{Emb}(P,G)|\) using the double sum in (11). The total time is bounded by (9) since the embeddings \(\varphi_j\) and indices \(\rho, C_j, X\) can both be represented using \(O(k)\) words of \(O(\log n)\) bits.

\section{The main algorithm}

This section proves Theorem 2. Let \(P\) be a connected \(k\)-vertex graph with a vertex balancer of size \(b\) and let \(G\) be an \(n\)-vertex graph.

First, let us observe that we have trivially \(|\text{Sub}(P,G)| = 0\) unless \(\Delta(P) \leq \Delta(G)\). Furthermore, we observe that \(\text{Aut}(P) = \text{Emb}(P,P)\).
The main algorithm starts by verifying that both \( k \leq n \) and \( \Delta(P) \leq \Delta(G) \); if this is not the case, the algorithm gives the output 0 and stops.

Next, the algorithm computes \( |\text{Aut}(P)| = |\text{Emb}(P,G)| \) using the algorithm in Lemma 6 with \( G \) set to equal \( P \). Since \( k \leq n \) and \( \Delta(P) \leq \Delta(G) \), it is immediate from (9) that this computation of \( |\text{Aut}(P)| \) runs within the main time bound (2).

Finally, the algorithm computes \( |\text{Emb}(P,G)| \) using the algorithm in Lemma 6 and, using (3), gives the output
\[
|\text{Sub}(G,P)| = \frac{|\text{Emb}(P,G)|}{|\text{Aut}(P)|}.
\]
Since (9) is bounded by (2), the total running time is bounded by (2). This completes the proof of Theorem 2.

## 7 Corollaries for elementary connected graphs

This section establishes Corollary 3. We start with a straightforward lemma on balancers.

\begin{lemma}
\[ k\text{-vertex path admits a balancer of size } 2 - (k \mod 2).\]
\[ k\text{-vertex cycle admits a balancer of size } 2 + (k \mod 2).\]
\[ k\text{-vertex tree for } k \geq 3 \text{ admits a balancer of size at most } \lfloor k/3 \rfloor.\]
\end{lemma}

\begin{proof}
A \( k \)-vertex path \( v_1, \ldots, v_k \) contains the balancer \( \{ v_{[k/2]} \} \) for odd \( k \) and the balancer \( \{ v_{[k/2]}, v_{[k/2]+1} \} \) for even \( k \). A \( k \)-vertex cycle \( v_1, \ldots, v_k \) for \( k \geq 2 \) contains the balancer \( \{ v_1, v_{[k/2]} \} \) for even \( k \) and the balancer \( \{ v_1, v_{[k/2]}, v_k \} \) for odd \( k \).

We turn to the third item. Every \( k \)-vertex tree contains a centroid vertex \( c \), which cuts it into subtrees \( T_1, \ldots, T_r \) of size \( k_1, \ldots, k_r \) with \( k_1 \leq k/2 \), \( \sum_i k_i = k - 1 \), and \( r \geq 2 \). Put the largest two subtrees, say \( T_1 \) and \( T_2 \), into \( C_1 \) and \( C_2 \), respectively, and add \( c \) to each. If \( r = 2 \) then \( k_1 = k_2 = (k - 1)/2 \) and \( c \) itself is a singleton balancer. Otherwise, add each of the remaining \( r - 2 \) trees, smallest first, to \( C_1 \) or \( C_2 \) such that \( |C_1| \) and \( |C_2| \) both remain at most \((k - 1)/2\). This process continues until the last tree, say \( T_3 \), which is instead added to both \( C_1 \) and \( C_2 \). If \( C_1 \) and \( C_2 \) now have unequal size, say \( |C_1| > |C_2| \) then repeatedly remove leaf nodes of \( T_3 \) from \( C_1 \) until \( |C_1| = |C_2| \). The resulting balancer \( C_1 \cap C_2 \) consists of \( c \) together with a subtree of \( T_3 \), so we have
\[
|C_1 \cap C_2| \leq k_3 + 1,
\]
and since
\[
k_3 \leq k_2 \leq k_1 \leq \frac{1}{3}(k - 1), \tag{12}
\]
so we see that the balancer is roughly \( \frac{1}{3}k \). For the precise bound in the lemma, the proceed by cases. If \( (k \mod 3) = 1 \) then \( \frac{1}{3}(k - 1) \) is an integer, so we can just write
\[
k_3 + 1 \leq \frac{1}{3}(k - 1) + 1 = [(k - 1)/3] + 1 = [k/3].
\]
If \( (k \mod 3) \in \{0, 2\} \) then the bound (12) cannot hold with equality, since otherwise the total number of vertices would be \( k_1 + k_2 + k_3 + 1 = 1 \) (mod 3). Thus,
\[
k_3 + 1 < \frac{1}{3}(k - 1) + 1 = \frac{1}{3}k + \frac{2}{3} \leq \lfloor k/3 \rfloor + 1.
\]
Since both sides of this strict inequality are integers, the bound in the lemma holds.
\end{proof}
Let us now proceed with a proof of Corollary 3. Recalling (2), for a connected $k$-vertex pattern $P$ with balancer size $b$, the main algorithm runs in time

$$O\left((2\Delta - 2)^{k + b - 1} - n k^2 \log n \right).$$

For a $k$-vertex path, we have $b = 2 - (k \mod 2) \leq 2$ by Lemma 7 and thus

$$\frac{k + b}{2} - 1 = \left\lfloor \frac{k}{2} \right\rfloor$$

implies the claimed running time

$$O\left((2\Delta - 2)^{\left\lfloor \frac{k}{2} \right\rfloor} n k^2 \log n \right).$$

For a $k$-vertex cycle, we have $b = 2 + (k \mod 2) \leq 3$ by Lemma 7 and thus

$$\frac{k + b}{2} - 1 = \left\lceil \frac{k}{2} \right\rceil$$

implies the claimed running time

$$O\left((2\Delta - 2)^{\left\lceil \frac{k}{2} \right\rceil} n k^2 \log n \right).$$

For a $k$-vertex tree with $k \geq 3$, we have $b \leq \left\lceil \frac{k}{3} \right\rceil$ by Lemma 7 and thus

$$\frac{k + b}{2} - 1 \leq \frac{k + \left\lceil \frac{k}{3} \right\rceil}{2} - 1 \leq \left\lceil \frac{2k - 3}{3} \right\rceil$$

implies the claimed running time

$$O\left((2\Delta - 2)^{\left\lceil \frac{2k - 3}{3} \right\rceil} n k^2 \log n \right).$$

This completes the proof of Corollary 3.

**Treewidth.** The notion of balancer is reminiscent of, but different from, the “balanced separators” that appear in the study of the graph parameter treewidth. However the latter is both more permissive and more strict, and no general corollaries for graphs of bounded treewidth follow from our results.

To see this, we exhibit infinite families of graphs where the two notions differ. One one hand, low treewidth is a global property that extends to all subgraphs. For instance, consider the $k$-vertex graph formed by connecting two $r$-cliques, where $r = \frac{1}{2}k - 1$, by identifying one vertex in each clique with the endpoints of a 3-vertex path. (This is the $r$-barbell graph with a subdivided bridge.) This graph has linear treewidth $\frac{1}{2}k - 2$, but admits a one-vertex balancer.

On the other hand, Definition 1 requires $C_1$ and $C_2$ to be connected, which the parts arising from a tree-decomposition need not be. For instance, consider the $k$-vertex graph $T$ formed by from three $r$-vertex paths, where $r = \frac{1}{3}(k - 1)$, by identifying one endpoint in each path with the leaves of the 4-vertex “Claw graph” $K_{1,3}$. This graph is a tree and thus has treewidth 1. Any partition of $V(T)$ into $C_1$ and $C_2$ must put at least two leaves into the same part, say $C_1$. Since $T[C_1]$ is connected, the unique path between these leaves must belong to $C_1$, so $|C_1| \geq 2r + 1$. By the balancing condition, $|C_2| = |C_1| \geq 2r + 1$. But then the balancer has size at least $|C_1 \cap C_2| = |C_1| + |C_2| - |C_1 \cup C_2| \geq 4r + 2 - (3r + 1) = r + 1 = \frac{1}{3}k + \frac{2}{3}$. (Note that this derivation matches the tree bound from Lemma 7, showing that neither construction can be improved.) We conclude that there is an infinite family of graphs of treewidth 1 whose balancers have size at least $\frac{1}{3}k$. 


A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto

References


Counting Connected Subgraphs with Maximum-Degree-Aware Sieving

Target Set Selection in Dense Graph Classes

Pavel Dvořák
Computer Science Institute, Charles University, Prague, Czech Republic
koblich@iuuk.mff.cuni.cz

Dušan Knop
Algorithmics and Computational Complexity, Faculty IV, TU Berlin, Berlin, Germany and
Faculty of Information Technology, Czech Technical University in Prague, Czech Republic
dusan.knop@gmail.com

Tomáš Toufar
Computer Science Institute, Charles University, Prague, Czech Republic
toufi@iuuk.mff.cuni.cz

Abstract
In this paper we study the Target Set Selection problem from a parameterized complexity perspective. Here for a given graph and a threshold for each vertex the task is to find a set of vertices (called a target set) to activate at the beginning which activates the whole graph during the following iterative process. A vertex outside the active set becomes active if the number of so far activated vertices in its neighborhood is at least its threshold.

We give two parameterized algorithms for a special case where each vertex has the threshold set to the half of its neighbors (the so called Majority Target Set Selection problem) for parameterizations by the neighborhood diversity and the twin cover number of the input graph.

We complement these results from the negative side. We give a hardness proof for the Majority Target Set Selection problem when parameterized by (a restriction of) the modular-width – a natural generalization of both previous structural parameters. We show that the Target Set Selection problem parameterized by the neighborhood diversity when there is no restriction on the thresholds is W[1]-hard.

2012 ACM Subject Classification Theory of computation → Parameterized complexity and exact algorithms

Keywords and phrases parameterized complexity, target set selection, dense graphs

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.18

Related Version arXiv:1610.07530

1 Introduction
We study the Target Set Selection problem (also called Dynamic Monopolies), using notation according to Kempe et al. [16], from parameterized complexity perspective. We use standard notions of parameterized complexity, see [9]. Let \( G = (V, E) \) be a graph, \( S \subseteq V \), and \( f : V \rightarrow \mathbb{N} \) be a threshold function. The activation process arising from the set \( S_0 = S \) is an iterative process with resulting sets \( S_0, S_1, \ldots \) such that for \( i \geq 0 \)

\[
S_{i+1} = S_i \cup \{v \in V : |N(v) \cap S_i| \geq f(v)\},
\]

The research leading to these results has received funding from the European Research Council under the European Union’s Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement n. 616787.

Part of this work has been done while affiliated with Department of Informatics, University of Bergen, Norway and supported by the project NFR MULTIVAL.
where by \(N(v)\) we denote the set of vertices adjacent to \(v\). Note that after at most \(n = |V|\) rounds the activation process has to stabilize – that is, \(S_i = S_{i+1}\) for all \(i > 0\). We say that the set \(S\) is a target set if \(S_i = V\) (for the activation process \(S = S_0, \ldots, S_n\)).

**Target Set Selection**

*Input:* A graph \(G = (V,E)\), \(f: V \rightarrow \mathbb{N}\), and a positive integer \(b \in \mathbb{N}\).

*Task:* Find a target set \(S \subseteq V\) of size at most \(b\) or report that there is no such set.

We call the input integer \(b\) the budget. The problem interpretation and computational complexity clearly may vary depending on the input function \(f\). There are three important settings studied (as we will discuss later) – namely constant, majority, and a general function. If the threshold function \(f\) is the majority (i.e., \(f(u) = \lceil\deg(u)/2\rceil\) for every vertex \(u \in V\)) we denote the problem as **Majority Target Set Selection**.

**Motivation.** The **Target Set Selection** problem was introduced by Domingos and Richardson [10] in order to study influence of direct marketing on a social network. It is noted therein that it captures e.g. viral marketing [21]. The **Target Set Selection** problem is important also from the graph theoretic viewpoint, since it generalizes many well known \(\text{NP}\)-hard problems on graphs. These problems include

\(=\) **Vertex Cover** [4] – set \(f(v) = \deg(v)\) for all \(v \in V\) and

\(=\) **Irreversible \(k\)-Conversion Set** [11], **\(k\)-Neighborhood Bootstrap Percolation** [2] – the **Target Set Selection** problem with all thresholds fixed to \(k\).

**Previous Results.** The **Target Set Selection** problem received attention of researchers in theoretical computer science in the past years. A general upper bound on the number of selected vertices under majority constraints is \(|V|/2 + 1\) [1]. The **Target Set Selection** problem admits an \(\text{FPT}\) algorithm when parameterized by the vertex cover number [19]. A \(t^{O(w)}\ poly(n)\) algorithm is known, where \(w\) is the tree-width of the input graph and \(t\) is an upper-bound on the threshold function [3], that is, \(f(v) \leq t\) for every vertex \(v\). This is essentially optimal, as the **Target Set Selection** problem parameterized by the path-width is \(W[1]\)-hard for majority [6] and general functions [3]. The **Target Set Selection** problem is solvable in linear time on trees [4] and more generally on block-cactus graphs [5]. The optimization variant of the **Target Set Selection** problem is hard to approximate [4] within a polylogarithmic factor. For more and less recent results we refer the reader to a survey by Peleg [20]. Cicalese et al. [8, 7], considered versions of the problem in which the number of rounds of the activation process is bounded. For graphs of bounded clique-width, given parameters \(a, b, \ell\), they gave polynomial-time algorithms to determine whether there exists a target set of size \(b\), such that at least \(a\) vertices are activated in at most \(\ell\) rounds. Recently Hartmann [15] gave a single-exponential \(\text{FPT}\) algorithm for **Target Set Selection** parameterized by clique width when all thresholds are bounded by a constant.

**Our Results.** In this work we generalize some results obtained by Nichterlein et al. [19]. Chopin et al. [6] essentially proved that in sparse graph classes (such as graphs with the bounded tree-width) parameterized complexity of the **Majority Target Set Selection** problem is the same as for the **Target Set Selection** problem. For these graph classes, it is not hard to see that e.g. if the threshold for vertex \(v\) is set above the majority (i.e., \(f(v) > \lceil\deg(v)/2\rceil\)), then we may add \(2f(v) - \lceil\deg(v)/2\rceil\) vertices neighboring with \(v\) only and the parameter stays unchanged whereas the threshold of \(v\) dropped to majority. However, this is not true in general for dense graph classes. We demonstrate this phenomenon for the parameterization by the neighborhood diversity. We show parameterized algorithm for
a function which generalizes both constant and majority threshold functions. We call this function uniform (and the corresponding problem **Uniform Target Set Selection**), see the next section for a proper definition. Roughly speaking all vertices belonging to a same part of a graph decomposition must possess the same value of the threshold function. In a slight contrast to the previous results, we derive an **FPT** algorithm that, instead of the maximal threshold value $t$, depends on the size of the image of the threshold function for graphs having bounded neighborhood diversity.

**Theorem 1.** There is an **FPT** algorithm for the **Uniform Target Set Selection** problem parameterized by the neighborhood diversity of the input graph.

**Theorem 2.** The **Target Set Selection** problem is **W[1]-hard** parameterized by the neighborhood diversity of the input graph.

The complexity of the **Majority Target Set Selection** problem is not resolved for parameterization by the cluster vertex deletion number [6] (the number of vertices whose removal from the graph results in a collection of disjoint cliques). We have a positive result for a slightly stronger parameterization: we assume that for every vertex we remove and every clique the vertex is either completely adjacent to the whole clique or is completely nonadjacent. This result also suggests that various weighted variants of the **Target Set Selection** problem may be in **FPT** when parameterized by the vertex cover number.

**Theorem 3.** There is an **FPT** algorithm for the **Uniform Target Set Selection** problem parameterized by the size of the twin cover of the input graph.

**Theorem 4.** The **Majority Target Set Selection** problem is **W[1]-hard** parameterized by the modular-width of the input graph.

### 2 Preliminaries on Structural Graph Parameters

We give formal definitions of several graph parameters used in this work. To get better acquainted with these parameters, we provide a map of the considered parameters in Figure 1.

For a graph $G = (V, E)$ the set $U \subseteq V$ is called a **vertex cover** of $G$ if for every edge $e \in E$ it holds that $e \cap U \neq \emptyset$. The **vertex cover number** of a graph, denoted as $\text{vc}(G)$, is the least integer $k$ for which there exists a vertex cover of size $k$.

As the vertex cover number is (usually) too restrictive, many authors focused on defining other (i.e., weaker) structural parameters. Three most well-known parameters of this kind are the path-width, the tree-width, and the clique-width. Classes of graphs with the bounded tree-width (even though only the tree-width is claimed). We show that this is already the case for parameterization by the (restricted) modular-width that generalizes both neighborhood diversity and twin cover number.

**Theorem 4.** The **Majority Target Set Selection** problem is **W[1]-hard** parameterized by the modular-width of the input graph.
Neighborhood Diversity. We say that two distinct vertices \( u, v \) are of the same neighborhood type if they share their respective neighborhoods, that is, when \( N(u) \setminus \{v\} = N(v) \setminus \{u\} \).

\[ \text{Definition 5} \text{(Neighborhood diversity [17])} \]

A graph \( G = (V, E) \) has neighborhood diversity at most \( w \) (\( nd(G) \leq w \)) if there exists a decomposition \( D_{nd} = (C_i)_{i=1}^w \) of \( V = C_1 \cup \cdots \cup C_w \) (we call the sets \( C_i \) types) such that all vertices in a type have the same neighborhood type.

Note that every type induces either a clique or an independent set in \( G \) and two types are either joined by a complete bipartite graph or no edge between vertices of the two types is present in \( G \). Thus, we use the notion of a type graph – that is a graph \( T_G \) representing the graph \( G \) and its neighborhood diversity decomposition in the following way. The vertices of type graph \( T_G \) are the types \( C_1, \ldots, C_w \) and two such vertices are joined by an edge if all the vertices of corresponding types are adjacent. We would like to point out that it is possible to compute the neighborhood diversity of a graph in linear time [17].

Twin Cover. If two vertices \( u, v \) have the same neighborhood type and \( e = \{u, v\} \) is an edge of the graph, we say that \( e \) is a twin edge.

\[ \text{Definition 6} \text{(Twin cover number [14])} \]

A set of vertices \( T \subseteq V \) is a twin cover of a graph \( G = (V, E) \) if for every edge \( e \in E \) either \( T \cap e \neq \emptyset \) or \( e \) is a twin edge. We say that \( G \) has twin cover number \( t \) (\( tc(G) = t \)) if the size of a minimum twin cover of \( G \) is \( t \).

Note that after removing \( T \) from a graph \( G \) the resulting graph consists of disjoint union of cliques – we call them twin cliques. Moreover, for every vertex \( v \) in \( T \) and a twin clique \( C \) holds that \( v \) is either adjacent to every vertex in \( C \) or to none of them. A twin cover decomposition \( D_{tc} = (C_i)_{i=1}^w \) of a graph \( G \) is a partition of \( V(G) \) such that each \( C_i \) is either a vertex of the twin cover or a twin clique.

Note that the twin cover number can be upper-bounded by the vertex cover number. The structure of graphs with bounded twin cover is very similar to the structure of graphs with bounded vertex cover number. Thus, there is a hope that many of known algorithms for graphs with bounded vertex cover number can be easily turned into algorithms for graphs with bounded twin cover number.

Uniform Threshold Function. As it is possible to compute the neighborhood diversity (or the twin cover) decomposition in polynomial time (or FPT-time, respectively), we may assume that the decomposition is given in the input. Given a decomposition \( D \) (\( D_{nd} \) or \( D_{tc} \)) a threshold function \( f : V(G) \rightarrow \mathbb{N} \) is uniform with respect to \( D \) if \( f(u) = f(v) \) for every \( u, v \in C \).
and every $C \in \mathcal{D}$. Observe that this notion generalizes the previously studied [6] model in which the threshold function is required to satisfy $f(u) = f(v)$ whenever $|N(u)| = |N(v)|$, since this indeed holds if $u, v \in C$ and $C \in \mathcal{D}$. It is not hard to see that the uniform function generalizes both the constant and the majority functions for the twin cover number and the neighborhood diversity; this is true already for the later notion.

Moreover, if $f(v)$ is bounded by a constant $c$ for all $v \in V(G)$, then there exists $\mathcal{D}_{nd}$ with $\|\mathcal{D}_{nd}\| \leq c \cdot nd(G)$ such that $f$ is uniform with respect to $\mathcal{D}_{nd}$. We stress here that this construction is not legal for the twin cover decompositions. Uniform Target Set Selection is a variant of Target Set Selection, where the input instance $(G, f, b, \mathcal{D})$ is restricted in such a way that the function $f$ is uniform with respect to $\mathcal{D}$.

**Modular-width.** Both the neighborhood diversity and the twin cover number are generalized by the modular-width. Here we deal with graphs created by an algebraic expression that uses the following four operations:
1. Create an isolated vertex.
2. The disjoint union of two graphs, that is from graphs $G = (V, E), H = (W, F)$ create a graph $(V \cup W, E \cup F)$.
3. The complete join of two graphs, that is from graphs $G = (V, E), H = (W, F)$ create a graph with vertex set $V \cup W$ and edge set $E \cup F \cup \{\{v, w\} : v \in V, w \in W\}$. Note that the edge set of the resulting graph can be also written as $E \cup F \cup (V \times W)$.
4. The substitution operation with respect to a template graph $T$ with vertex set $\{v_1, v_2, \ldots, v_k\}$ and graphs $G_1, G_2, \ldots, G_k$ created by an algebraic expression; here $G_i = (V_i, E_i)$ for $i = 1, 2, \ldots, k$. The substitution operation, denoted by $T(G_1, G_2, \ldots, G_k)$, results in the graph on vertex set $V = V_1 \cup V_2 \cup \cdots \cup V_k$ and edge set

$$E = E_1 \cup E_2 \cup \cdots \cup E_k \cup \bigcup_{\{u, v\} \in E(T)} \{\{u, v\} : u \in V_i, v \in V_j\}.$$  

**Definition 7 (Modular-width [13]).** Let $A$ be an algebraic expression that uses only operations 1–4 above. The width of the expression $A$ is the maximum number of operands used by any occurrence of operation 4 in $A$. The modular-width of a graph $G$, denoted $mw(G)$, is the least positive integer $k$ such that $G$ can be obtained from such an algebraic expression of width at most $k$.

An algebraic expression of width $mw(G)$ can be computed in linear time [22].

**Restricted Modular-width.** We would like to introduce here a restriction of the modular-width that still generalizes both the neighborhood diversity and the twin cover number. The algebraic expression used to define a graph $G$ may contain the substitution operation at most once and if it contains the substitution operation it has to be the last operation in the expression. However, there is no limitation for the use of operations 1–3.

### 3 Positive Results

In this section we give proofs of Theorem 1 and 3. In the first part we discuss the crucial property of dense structural parameters – the uniformity of neighborhoods. This, opposed to e.g. the cluster vertex deletion number, allows us to design parameterized algorithms.

In this section by a decomposition $\mathcal{D}$ we mean a neighborhood diversity or a twin cover decomposition.
Lemma 8. Let \( G = (V, E) \) be a graph, \( D \) be a decomposition of \( G \), \( S \subseteq V \) be a target set, and \( f \) be a uniform threshold function with respect to \( D \). Let \( G, f, b, D_c \) be an activation process arising from \( S \). For each round \( i \in \mathbb{N}_0 \) one of the following holds either
1. \( S_i \cap C = S_0 \cap C \), or
2. \( S_i \cap C = C \).

Moreover, there exist \( j \) with \( j \in \mathbb{N}_0 \) such that for \( C \) the first item applies in rounds \( 0, \ldots, j \) and the second in rounds \( j+1, \ldots \).

Proof. Since \( f \) is uniform, it is constant on \( S \). The proof is by induction on the round number \( i \). The statement clearly holds for \( i = 0 \). Suppose that the statement is valid for all \( i' < i \) but not for \( i \), that is, in the \( i \)-th round there are two vertices \( u, v \in C \) such that \( u \in S_i \setminus S_{i-1} \) and \( v \notin S_i \). This is impossible, as both \( u \) and \( v \) have the same neighborhood type and \( f(u) = f(v) \). Thus if \( u \) gets activated, then \( v \) must be activated as well. The “moreover” part follows from the monotonicity of the activation process \((S_i \subseteq S_{i+1})\).

Let \( C \in D \). For a threshold function \( f \) which is constant on \( C \) we define \( f'(C) \) as \( f(v) \) for arbitrary vertex \( v \) in \( C \). By Lemma 8, we say that \( C \) is activated in a round \( i \) if \( S_i \cap C = C \) and \( S_j \cap C = S_0 \cap C \) for every \( j < i \). We denote \( a_i^g(v) \) the number \( |S_{i-1} \cap N(v)| \), i.e., the number of active neighbors of \( v \) in the round \( i \) in the activation process arising from the set \( S \). Thus, a vertex \( v \) is activated in the first round \( i \) when \( a_i^g(v) \geq f(v) \) holds.

3.1 Uniformity and Twin Cover

In this subsection we present an algorithm for Uniform Target Set Selection parameterized by the twin cover number.

Trivial Bounds on the Minimum Target Set. Let \( G = (V, E) \) be a graph with twin cover \( T \) of size \( t \) and let \( C_1, C_2, \ldots, C_q \) be the twin cliques of \( G \). For a twin clique \( C \) by \( N(C) \) we denote the common twin cover neighborhood, that is, \( N(v) \cap T \) for any \( v \in C \). We show that there is a small number of possibilities how the optimal target set can look like. Let \( b_C = \max(f'(C) - |N(C)|, 0) \) for a twin clique \( C \).

Observation 9. If the minimum target set of \( G \) has size \( s \), then \( B \leq s \leq B + t \) for \( B = \sum_{i=1}^{q} b_C \).

Proof. Let \( S \) be a target set for \( G \) of size \( s \). Suppose there is a twin clique \( C \) such that \( |S \cap C| = p < b_C \). It means that \( b_C > 0 \). Let \( v \in C \setminus S \). Note that \( p < |C| \), thus such a vertex \( v \) exists. For the vertex \( v \) it holds that \( a_i^g(v) < p + |N(C)| \) for every round \( i \) of the process. Thus, the vertex \( v \) is never activated because \( p + |N(C)| < b_C + |N(C)| = f'(C) \) and \( S \) is not a target set. On the other hand, if we put \( b_C \) vertices from each twin clique \( C \) into a set \( S' \), then the set \( S' \cup T \) is a target set because every vertex not in \( S' \) is activated in the first round.

Structure of the Solution. Let \( (G, f, b, D_c) \) be an instance of Uniform Target Set Selection with \( tc(G) = t \). By Observation 9, if \( b < \sum b_C \), then we automatically reject. On the other hand, if \( b \geq t + \sum b_C \), then we automatically accept. Let \( w = b - \sum b_C \). Thus, to find a target set of size \( b \) we need to select \( w \) excess vertices from the twin cliques and the twin cover. We will show there are at most \( g(t) \) interesting choices for these \( w \) excess vertices for some computable function \( g \) and those choices can be efficiently generated. Since we can check if a given set \( S \subseteq V(G) \) is a target set in polynomial time, there is an FPT-algorithm for Uniform Target Set Selection.
We start with an easy preprocessing. Let $C$ be a twin clique with $b_C > 0$. We select $b_C$ vertices $V' \subseteq C$ and remove them from the graph $G$. We also decrease the threshold value by $b_C$ of every vertex which was adjacent to $V'$ (recall that vertices in $V'$ have the same neighborhood type, thus any vertex adjacent to some vertex in $V'$ is adjacent to all vertices in $V'$). Formally, we get an equivalent instance $(G_1, f_1, b - b_C, D_{tc}')$, where $G_1$ is $G$ without vertices $V'$, $D_{tc}'$ is $D_{tc}$ restricted to $V(G_1)$ and

$$f_1(v) = \begin{cases} f(v) & v \notin N_G(V') \\ f(v) - b_C & v \in N_G(V'). \end{cases}$$

It is easy to see that the instances $(G, f, b, D_{tc})$ and $(G_1, f_1, b - b_C, D_{tc}')$ are equivalent, because any target set of $G$ needs at least $b_C$ vertices in the twin clique $C$ due to Observation 9. Note that the function $f_1$ is uniform with respect to $D_{tc}'$. We repeat this process for all twin cliques. From now on we suppose that the instance $(G, f, b, D_{tc})$ is already preprocessed. Thus, for every twin clique $C$ it holds that $b_C = 0$ and $f'(C) \leq N(C) \leq t$.

We say that a twin clique $C$ is of a type $(Q, r)$ for $Q \subseteq T, r \leq t$ if $Q = N(C)$ and $f'(C) = r$. Two twin cliques $C$ and $D$ are of the same type if $N(C) = N(D)$ and $f'(C) = f'(D)$. Note that there are at most $(t + 1) \cdot 2^l$ distinct types of the twin cliques.

We start to create a possible target set $S$ of size $b$. We add $w_1$ (for some $w_1 \leq w$) vertices from the twin cover $T$ to $S$ (there are at most $2^l$ such choices). Now we need to select $w_2 = w - w_1$ excess vertices from twin cliques to $S$.

The number of the twin cliques of one type may be large. Thus, for the twin cliques we need some more clever way than try all possibilities. The intuition is that if we want to select some excess vertices from a clique of a type $(Q, r)$ it is a “better” choice to select the vertices from large cliques of the type $(Q, r)$. We assign to each type $(Q, r)$ a number $w_{(Q,r)}$ how many excess vertices would be in twin cliques of type $(Q, r)$. We prove that it suffices to distribute $w_{(Q,r)}$ excess vertices among the $w_{(Q,r)}$ largest twin cliques of the type $(Q, r)$.

**Definition 10.** Let $C_1, \ldots, C_p$ be all twin cliques of type $(Q, r)$ ordered by the size in a descending order, i.e., for all $i < p$ holds that $|C_i| \geq |C_{i+1}|$. We say that a target set has a hole $(C_i, C_j)$ for $i < j$ if $|S \cap C_i| = 0$ and $|S \cap C_j| \geq 1$. A target set is $(Q, r)$-leaky if it has a hole and it is $(Q, r)$-compact otherwise.

Our goal is to prove that if there is a target set $S$ which is $(Q, r)$-leaky, then there is also a target set $R$ which is $(Q, r)$-compact and $|R| = |S|$.

**Lemma 11.** Suppose there is a target set $S$ for a graph $G$ with a threshold function $f$ and $S$ is $(Q, r)$-leaky for some twin clique type $(Q, r)$. Then, there is a target set $R$ such that:

1. It holds that $|R| = |S|$.
2. The sets $R$ and $S$ differ only at the twin cliques of the type $(Q, r)$.
3. The set $R$ is $(Q, r)$-compact.

**Proof Sketch.** Let set $S$ has a hole $(C_i, C_j)$ for the twin cliques of the type $(Q, r)$. We create a target set $R$ by removing vertices from $C_j$ and adding the same number of vertices from $C_i$. Formally, $R = (S \setminus C_j) \cup X$, where $X \subseteq C_i$ and $|X| = |S \cap C_j|$. The verification that $R$ is a target set is technical but rather straightforward. The whole proof is in the full version of the paper.

If we repeat Lemma 11 for every type $(Q, r)$, we get a target set without any hole. To summarize how to distribute $w$ excess vertices:
Target Set Selection in Dense Graph Classes

1. Pick $w_1$ vertices from the twin cover $T$, in total $2^t$ choices.
2. Distribute $w_2 = w - w_1$ excess vertices among $t \cdot 2^t$ types of twin cliques, in total $(t \cdot 2^t)^t = 2^{O(t^3)}$ choices.
3. Distribute $w_{(Q,r)}$ excess vertices among the $w_{(Q,r)}$ largest cliques of type $(Q,r)$, in total $t^t$ choices.

By this we create $2^{O(t^2)}$ candidates for a target set. For each candidate we test whether it is a target set for $G$ or not. If any candidate is a target set, then we find a target set of size $b$. If no candidate is a target set, then by argumentation above we know the graph $G$ has no target set of size $b$. This finishes the proof of Theorem 3.

3.2 Neighborhood diversity

In this section we prove that the Uniform Target Set Selection problem admits an FPT algorithm on graphs of the bounded neighborhood diversity. We again use Lemma 8. Note that in each round of the activation process at least one type has to be activated. This implies that there are at most $nd(G)$ rounds of the activation process. We use this fact to model the whole activation process as an integer linear program which is then solved using Lenstra’s celebrated result:

\[ \text{Proposition 12 ([18, 12])}. \text{ Let } p \text{ be the number of integral variables in a mixed integer linear program and let } L \text{ be the number of bits needed to encode the program. Then it is possible to find an optimal solution in time } O(p^{2.5p}\text{poly}(L)) \text{ and a space polynomial in } L. \]

There has to be an order in which the types are activated in order to activate whole graph. Since there are $t = nd(G)$ types, we can try all such orderings. Let us fix an order $\prec$ on types. To construct the ILP we further need to know which types are fully activated at the beginning. Denote by $c_0$ the number of such types. Once the order $\prec$ is fixed the set of fully activated types at the beginning is determined by $c_0$. Since $c_0$ can attain values $0, \ldots, t$ we can try all $t + 1$ possibilities. Now, with both $\prec$ and $c_0$ fixed, denote the set of the types activated in the beginning by $\mathcal{T}_0$.

Observe further that, as the vertices in a type share all neighbors, the only thing that matters is the number of activated vertices in each type and not the actual vertices activated. Thus, we have variables $x_C$ which corresponds to the number of vertices in type $C$ selected into a target set $S$.

Let $C$ be a type and $n_C$ be the number of vertices in $C$. Since we know when $C$ is activated, we know how many active vertices are in $C$ in each round. There are $x_C$ vertices before the activation of $C$ and $n_C$ after the activation. To formulate the integer linear program we denote the set of types by $\mathcal{T}$ and we write $D \in N(C)$ if the two corresponding vertices in the type graph $T_G$ are connected by an edge.

**ILP Formulation.**

\[
\begin{align*}
\text{minimize} & \quad \sum_{C \in \mathcal{T}} x_C \\
\text{subject to} & \quad f'(C) \leq \sum_{D \prec C, D \in N(C)} n_D + \sum_{D \succ C, D \in N(C)} x_D + [C \text{ is a clique}]x_C \quad \forall C \in \mathcal{T} \setminus \mathcal{T}_0 \\
\text{where} & \quad 0 \leq x_C < n_C \\
& \quad x_C = n_C \quad \forall C \in \mathcal{T} \setminus \mathcal{T}_0 \\
& \quad \forall C \in \mathcal{T}_0
\end{align*}
\]
As there are at most \( t! \) orders of the set \( T \) and \( t + 1 \) choices of \( c_0 \) this implies that the \textsc{Uniform Target Set Selection} problem can be solved in time \((t + 1)!^{O(t)} \text{poly}(n) = t^{O(t)} \text{poly}(n)\). Thus, we have proven Theorem 1.

## 4 Hardness Reductions

In this section we prove that \textsc{Target Set Selection} is \textsc{W[1]}-hard on graphs of the bounded neighborhood diversity for a general threshold function. We use an FPT-reduction from \( k\text{-Multicolored Clique} \).

### \( k\text{-Multicolored Clique} \)

**Parameter:** \( k \)

**Input:** A \( k \)-partite graph \( G = (V_1 \cup \cdots \cup V_k, E) \), where \( V_c \) is an independent set for every \( c \in [k] \) and they are pairwise disjoint.

**Task:** Find a clique of the size \( k \).

Let \( G \) be an input of \( k\text{-Multicolored Clique} \). We refer to a set \( V_c \) as to a \textit{color class} of \( G \) and to a set \( E_{cd} \) as to edges between color classes \( V_c \) and \( V_d \). The problem is \textsc{W[1]}-hard \cite{FPT} even if every color class \( V_c \) has the same size and the number of edges between every \( V_c \) and \( V_d \) is the same. For an easier notation, we denote the size of an arbitrary color class \( V_c \) by \( n + 1 \) and the size of an arbitrary set \( E_{cd} \) by \( m + 1 \). We describe a reduction from the graph \( G \) to an instance of \textsc{Target Set Selection} \( (G', f : V \rightarrow \mathbb{N}, b) \) where \( nd(G') = \mathcal{O}(k^2) \). The reduction runs in time \( \text{poly}(|G|) \). The graph \( G \) has a clique of size \( k \) if and only if the graph \( G' \) has a target set of size \( b \).

In the \( k\text{-Multicolored Clique} \) problem we need to select exactly one vertex from each color class \( V_c \) and exactly one edge from each set \( E_{cd} \). Moreover, we have to make certain that if \( \{u, v\} \in E_{cd} \) is a selected edge, then \( u \in V_c \) and \( v \in V_d \) are selected vertices.

**An Overview of Proof of Theorem 2.** We present a way of encoding a vertex \( v \) in a color class \( V_c \) of the graph \( G \) by two numbers \( v\text{-pos} \) and \( v\text{-neg} \) with \( v\text{-pos} + v\text{-neg} = n \). We proceed with encoding of edges similarly, however, edges are encoded by multiples of sufficiently large number \( q \). This we do in such a way that sum of the encoding of a vertex and an incident edge is unique. We create three types of gadgets: for selection vertices, for selection edges, and gadgets which check that the selected vertices are incident to the selected edges. Proofs of lemmas and theorems in this section are quite technical and they are presented in the full version of this paper.

### 4.1 Proof of Theorem 2

In order to present the hardness reduction we have to introduce some gadgets. We denote the name of a gadget by capital letters and we write parameters of the gadget into parentheses (e.g. \( L(s) \)). When speaking about concrete instance of a gadget kind \( L(s) \) we add a subscript, i.e., \( L_c(s) \). We omit parameters of the gadget it they are clear from the context.

**Selection Gadget.** First, we describe gadgets of the graph \( G' \) for selecting vertices and edges of the graph \( G \). For an overview of the reduction see Figure 2. The gadget \( L(s) \) is formed by two types \( L\text{-neg} \) and \( L\text{-pos} \) of equal size \( s \) (the number \( s \) will be determined later); we refer to these two types as the \textit{selection part}. For a vertex \( v \) in the selection part we set the value \( f(v) \) of the threshold to the degree of \( v \). It means that if some vertex \( v \) from the selection part is not selected into the target set, then all neighbors of \( v \) have to be active before the vertex \( v \) can be activated by the activation process. The selection gadget \( L \) is connected to the rest of the graph using only vertices from the selection part.
The last part of the gadget $L$ is formed by type $L$-guard of $s + 1$ vertices connected to both types in the selection part. For each vertex $v$ in $L$-guard type we set $f(v) = s$.

**Numeration of Vertices and Edges.** Now, we describe how we use the selection gadget. Let $V_c = \{v_0, \ldots, v_n\}$. For every color class $V_c$ we create a selection gadget $L_c = L(n)$. We select a vertex $v_i \in V_c$ to the multicolor clique if $i$ vertices in the $L_c$-pos type and $n - i$ vertices in the $L_c$-neg type of the gadget $L_c$ are selected into the target set.

The selection of edges is similar, however, a bit more complicated. Let $q \in \mathbb{N}$ and $E_{cd} = \{e_0, \ldots, e_m\}$. For every set $E_{cd}$ we create a selection gadget $L_{cd}$ of kind $L(qm)$. We select an edge $e_j \in E_{cd}$ to the multicolor clique if $q_j$ vertices in the $L_{cd}$-pos type of the gadget $L_{cd}$ are selected into the target set (and $q(m - j)$ vertices in the $L_{cd}$-neg are selected into the target set). Suppose $s$ vertices in the $L_{cd}$-pos type are selected into the target set. If $s$ is not divisible by $q$, then it is an invalid selection. We introduce a new gadget which controls that $s$ has to be divisible by $q$.

**Multiple Gadget.** A multiple gadget $M(q, s)$ consists of a selection gadget $L(qs)$ and 3 other types: $M$-pos, $M$-neg of $s$ vertices and $M$-guard of $qs$ vertices. The type $M$-pos is connected to the type $L$-pos and the type $M$-neg is connected to the type $L$-neg. The type $M$-guard is connected to the types $M$-pos and $M$-neg. Still, the rest of graph $G'$ is connected only to types $L$-pos and $L$-neg. Let $\{u_1, \ldots, u_s\}$ and $\{w_1, \ldots, w_s\}$ be vertices in $M$-pos type and $M$-neg type, respectively. We set thresholds $f(u_i) = f(w_i) = q_i$. For each vertex $v$ in $M$-guard we set $f(v) = s$.

**Incidence Gadget.** So far we described how we encode in graph $G'$ selecting vertices and edges to multicolor clique. It remains to describe how we encode the correct selection, i.e., if $v \in V_c$ and $e \in E_{cd}$ are selected vertex and edge to multicolor clique, then $v \in e$. We create $L_c(n)$ selection gadget for a color class $V_c$. We set the number $q$ to $n^2$ and create a multiple gadget $M_{cd}$ of kind $M(n^2, m)$ (with selection gadget $L_{cd}$) for a set $E_{cd}$. We join gadgets $L_c$ and $M_{cd}$ through an incidence gadget $I_{cd}$. The incidence gadget $I_{cd}$ has three types $I_{cd}$-pos and $I_{cd}$-neg of $m + 1$ vertices and $I_{cd}$-guard of $n + n^2m$ vertices. We connect the $I_{cd}$-guard type to the types $I_{cd}$-pos and $I_{cd}$-neg. Furthermore, we connect the type $I_{cd}$-pos to the types $L_c$-pos and $L_{cd}$-pos. Similarly, we connect the type $I_{cd}$-neg to the types $L_c$-neg and $L_{cd}$-neg.

We set thresholds of all vertices in the $I_{cd}$-guard type to $m + 2$. Recall there are $m + 1$ edges in the set $E_{cd}$. Thus, we can associate edges in $E_{cd}$ with vertices in $I_{cd}$-pos ($I_{cd}$-neg respectively) one-to-one. I.e., $V(I_{cd}$-pos) = $\{u_t : e_t \in E_{cd}\}$ and $V(I_{cd}$-neg) = $\{w_t : e_t \in E_{cd}\}$. Let $v_i \in V_c, e_j \in E_{cd}$ and $v_i \in e_j$. Recall that selecting $v_i$ and $e_j$ into a multicolor clique is encoded as selecting $i$ vertices in $L_c$-pos type and $n^2j$ vertices in $L_{cd}$-pos type into a target set. We set threshold of $u_j$ to $i + n^2j$ and threshold of $w_j$ to the “opposite” value $n - i + n^2(m - j)$.

Since we set the coefficient $q$ to $n^2$, for each edge $e_j \in E_{cd}$ and each vertex $v_i \in V_c$ the sum $i + n^2j$ is unique. Thus, every vertex in $I_{cd}$-pos ($I_{cd}$-neg) has a unique threshold. We will use this number to check the incidence.

**Reduction Correctness.** We described how from the graph $G$ with $k$ color classes (instance of $k$-MULTICOLORED CLIQUE) we create the graph $G'$ with the threshold function $f$ (input for TARGET SET SELECTION). For every color class $V_c$ we create a selection gadget $L_c$. For
every edge set $E_{cd}$ we create a multiple gadget $M_{cd}$. We join the gadgets $L_c$ and $M_{cd}$ by an incidence gadget $I_{cd}$ (gadgets $L_d$ and $M_{cd}$ are joint by a gadget $I_{cd}$). It is easy to see the following observations by constructions of $G'$.

- **Observation 13.** The graph $G'$ has polynomial size in the size of the graph $G$.

- **Observation 14.** Neighborhood diversity of the graph $G'$ is $O(k^2)$.

To finish the construction of an instance of TARGET SET SELECTION, we set the budget $b$ to $kn + (\frac{k}{2})n^2m$. The main idea of proofs of the following theorems is that we select a vertex $v_i \in V_c$ (or an edge $e_j \in E_{cd}$) into a clique if and only if we select $i$ vertices from the $L_{c\text{-pos}}$ type (or $n^2j$ vertices from the $L_{cd\text{-pos}}$ type). Theorem 2 is a corollary of Observation 13, 14 and the following theorem.

- **Theorem 15.** The graph $G$ contains a clique of size $k$ if and only if the graph $G'$ with the threshold function $f$ contains a target set of size $b$.

### 4.2 Overview of Proof of Theorem 4

In fact this can be seen as a clever twist of the ideas contained in the proof of Theorem 2. There are some nodes of the neighborhood diversity decomposition already operating in the majority mode – e.g. guard vertices – these we keep untouched. For vertices with threshold set to their degree one has to “double” the number of vertices in the neighborhood. Finally, one has to deal with types having different thresholds for each of its vertices, which is quite technical.

### 5 Conclusions

We have generalized ideas of previous works [3, 19] for the TARGET SET SELECTION problem. The presented results give a new idea how to encode selecting vertices and edges in the $k$-MULTICOLORED CLIQUE problem for showing $W[1]$-hardness. In particular, only few problems are known to be $W[1]$-hard when parameterized by neighborhood diversity – which is the case for the TARGET SET SELECTION problem.

Thus, we would like to address an open problem regarding structural parameterizations of the TARGET SET SELECTION problem. Determine parameterized complexity of the TARGET SET SELECTION problem parameterized by twin cover number. Furthermore, we are not
aware of other positive results concerning the number of different thresholds instead of the threshold upper-bound.

We would like to point out that in our proofs of \( \text{W}[1] \)-hardness the activation process terminates after constant number of rounds (independent of the parameter value and the size of the input graph). This is true also for all reductions given by Chopin et al. [6].

References


Counting Shortest Two Disjoint Paths in Cubic Planar Graphs with an NC Algorithm

Andreas Björklund
Department of Computer Science, Lund University, Sweden

Thore Husfeldt
BARC, IT University of Copenhagen, Denmark and Lund University, Sweden

Abstract
Given an undirected graph and two disjoint vertex pairs \(s_1, t_1\) and \(s_2, t_2\), the Shortest two disjoint paths problem (S2DP) asks for the minimum total length of two vertex disjoint paths connecting \(s_1\) with \(t_1\), and \(s_2\) with \(t_2\), respectively.

We show that for cubic planar graphs there are NC algorithms, uniform circuits of polynomial size and polylogarithmic depth, that compute the S2DP and moreover also output the number of such minimum length path pairs.

Previously, to the best of our knowledge, no deterministic polynomial time algorithm was known for S2DP in cubic planar graphs with arbitrary placement of the terminals. In contrast, the randomized polynomial time algorithm by Björklund and Husfeldt, ICALP 2014, for general graphs is much slower, is serial in nature, and cannot count the solutions.

Our results are built on an approach by Hirai and Namba, Algorithmica 2017, for a generalisation of S2DP, and fast algorithms for counting perfect matchings in planar graphs.

2012 ACM Subject Classification
Mathematics of computing → Paths and connectivity problems, Mathematics of computing → Combinatorial algorithms

Keywords and phrases Shortest disjoint paths, Cubic planar graph

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.19

Funding This work was supported by Swedish Research Council grant VR-2016-03855 “Algebraic Graph Algorithms” and Villum Foundation grant 16582.

Acknowledgements We thank Radu Curticapean for making us aware of the fast algorithms for determinants of matrices with a planar structure.

1 Introduction

Shortest disjoint \(A,B\)-paths, introduced by Hirai and Namba [12], is the following problem: Let \(G\) be an undirected graph with two non-empty disjoint vertex subsets \(A, B \subseteq V(G)\) of even size and an edge length function \(\ell: E(G) \to \{1, \ldots, L\}\). An edge subset \(E' \subseteq E(G)\) is a solution to Disjoint \(A,B\)-paths if it consists of \(\frac{1}{2}(|A| + |B|)\) disjoint paths with endpoints both in \(A\) or both in \(B\). The length \(\ell(E')\) of a solution is \(\sum_{e \in E'} \ell(e)\), and a shortest solution has length \(\ell_{A,B} = \min_{E'} \ell(E')\). The objective is to compute \(\ell_{A,B}\). The special case \(|A| = |B| = 2\) is a well-studied problem called Shortest two disjoint paths.

We write \(S_{A,B}\) for the number of solutions of length \(\ell_{A,B}\). A graph is cubic (sometimes called 3-regular) if every vertex has degree 3. We prove the following:
Counting Shortest Two Disjoint Paths in Cubic Planar Graphs

Figure 1: A solution of minimum length $\ell_{A,B} = 11$ to Shortest disjoint $A, B$-paths with $A = \{a_1, a_2\}$ and $B = \{b_1, b_2\}$. Since $|A| = |B| = 2$, this is also an example of Shortest two disjoint paths. Note that neither path is a shortest path between its terminals.

Theorem 1. For any cubic planar $n$-vertex graph $G$, disjoint vertex subsets $A$ and $B$, and edge length function $\ell : E(G) \to \{1, \ldots, L\}$, we can compute $\ell_{A,B}$ and $S_{A,B}$ in deterministic $\tilde{O}(2^{\omega/2}n^{\omega/2+2}L^2)$ time, where $\omega < 2.373$ is the exponent of square matrix multiplication.

In particular, for $|A| + |B| = O(1)$, the algorithm runs in deterministic time $\tilde{O}(n^{\omega/2+2}L^2)$.

To the best of our knowledge, no polynomial-time deterministic algorithm was known even for $|A| = |B| = 2$. Hirai and Namba’s algorithm [12] works for general graphs in randomized time $n^{O(|A\cup B|)}$, so Theorem 1 also shows that cubic planar graphs allow better exponential dependency on $|A \cup B|$. In the appendix we show that all our algorithms extend to the case where the graph has maximum degree 3.

Because we can count the solutions we can use well-known techniques to retrieve a witness for the shortest length. By using our algorithm as a subroutine, we can retrieve the $i$th witness in a lexicographical order of the solutions by a polynomial overhead self reduction, by peeling off edges one at a time and remeasuring the number of solutions. In particular, by choosing $i$ uniformly from $\{1, \ldots, S_{A,B}\}$, we can sample uniformly over the solutions without first explicitly constructing the list of solutions.

Our algorithm is based on counting perfect matchings in a planar graph. Vazirani [30] showed how every bit in the number of perfect matchings in a planar graph can be decided by an NC algorithm, i.e., uniform polylogarithmically shallow polynomial size circuits, an observation he attributes to Luby. Using his algorithm as a subroutine, we arrive at an efficient parallel algorithm. We state the result for Shortest two disjoint paths:

Theorem 2. For any cubic planar $n$-vertex graph $G$, disjoint vertex subsets $A$ and $B$ with $|A| = |B| = 2$, and edge length function $\ell : E(G) \to \{1, \ldots, L\}$, we can compute $\ell_{A,B}$ and $S_{A,B}$ by an NC algorithm.

The same statement holds as long as $|A| + |B|$ is logarithmic in $n$.

Via the Isolation lemma of Mulmuley et al. [23] we can also obtain a witness, i.e., a solution $E'$ of length $\ell(E') = \ell_{A,B}$, with a randomized NC algorithm. We note that the recent breakthrough result showing how to find a perfect matching in a planar graph in NC by Anari and Vazirani [2] doesn’t seem to be directly applicable to our problem. Our algorithm counts the solutions to Shortest two disjoint paths by an annihilation sieve, i.e. the number of solutions is an alternating sum of perfect matchings in a set of graphs, but many of the terms cancel each other. Hence there are many perfect matchings that do not correspond to a solution. Finding one deterministically won’t help us.

We also provide evidence that the exponential dependence on $|A| + |B|$ is necessary:

1 The $\tilde{O}(f(n))$ notation suppresses factors polylogarithmic in $f(n)$. 

Theorem 3. It is \#P-hard to simultaneously compute the length and the number of solutions to Shortest disjoint $A, B$-paths in cubic planar graphs.

1.1 Hirai and Namba’s Result

Hirai and Namba [12] shows that Shortest disjoint $A, B$-paths has a randomized algorithm running in $n^{O(|A \cup B|)}$ time, that w.h.p. finds the length of the shortest disjoint paths. Their algorithm is inspired by the algorithm of Gallai [9] that can be used to address the special case $B = \emptyset$, and the algorithm by Björklund and Husfeldt [4] for the special case $|A| = |B| = 2$. They apply a two-step method: First, expand $G$ into another edge weighted graph $G'$ using so-called Gallai paths, in a way that the weighted perfect matchings in $G'$ can be used to obtain the solution to the original problem. Then it uses the fact that counting perfect matchings in $G'$ modulo $2^k$ has a $n^{O(k)}$ time algorithm. In their reduction, each solution is counted $2^{(|A \cup B|/2)}$ times, so they need to set $k > \frac{1}{2}|A \cup B|$ to count something meaningful, but still small enough to keep the running time down. Thus, their algorithm is capable only of counting the solutions modulo a fixed small power of 2. The Isolation lemma [23] ensures that the number of solutions is not divisible by this small power of two. This is why [12] need randomness; the same problem appears in the shortest paths algorithm of [4].

1.2 Our Approach

We apply Hirai and Namba’s approach to the planar cubic case. It is well-known that in any planar graph we can count the perfect matchings in polynomial time. In particular we don’t just obtain the result modulo a small power of two. We use this, but there is one obstacle that needs to be addressed to accomplish this: The reduction Hirai and Namba use does not preserve planarity. Our contribution is to show that for cubic planar graphs, we can construct a set of $2^{(|A \cup B|/2)}$ cubic planar graphs, each having non-negative edge weights, so that a linear combination of the number of weighted perfect matchings in these graphs can be used to deduce the number of solutions to the Shortest disjoint $A, B$-paths in the original instance. We are inspired by the result of Galluccio and Loebl [10] that shows how to count perfect matchings in graphs of genus $g$ by constructing $4^g$ orientations and computing the Pfaffian for each of them. We choose to use a more direct approach instead of reducing to their result to make the description of our algorithm more self contained.

1.3 Related Work

Björklund and Husfeldt [4] showed that Shortest two disjoint paths in a general unweighted undirected graph has a polynomial time Monte Carlo algorithm. Colin de Verdière and Schrijver [8], and Kobayachi and Sommer [19] showed that for planar graphs, deterministic polynomial-time algorithms for the Shortest two disjoint paths exist if the four terminals lie on the boundary of at most two faces. The algorithm in the present paper works no matter where the terminals are, but is much slower. Still, it is significantly faster than the general $O(n^{11})$ time algorithm by Björklund and Husfeldt [4].

Very recently, Datta et al. [7] presented a deterministic algorithm independently of ours for Shortest $k$-disjoint paths in planar graphs conditioned on the terminals either all being placed on the same face or all source terminals on one face and the target terminals on another. In particular, restricted to the Shortest two disjoint path problem, their algorithm does not work for arbitrary placement of the terminals as ours does. Interestingly, their algorithm is also based on computing determinants just as ours and can count the solutions just as our algorithms can, although they don’t use Pfaffian orientations as we do.
For the decision problem of detecting two disjoint paths joining given vertex pairs, no matter their length, deterministic polynomial-time algorithms have been known since 1980 for general graphs, by Ohtsuki [24], Seymour [25], Shiloah [26], and Thomassen [28]; all published independently. Tholey [27] reduced the running time for that problem to near-linear. Khuller et al. [17] showed that the problem can be solved in NC.

The \( k \)-disjoint paths problem is the natural generalisation of the two disjoint paths problem: Given a list \( \{(s_1, t_1), \ldots, (s_k, t_k)\} \) of terminal pairs, decide if there exist \( k \) disjoint paths connecting \( s_i \) with \( t_i \) for \( i \in \{1, \ldots, k\} \). Again neglecting the length of the solution, this problem has a polynomial time algorithm in general graphs for fixed \( k \), but the dependence on \( k \) is horrible \((\exp \exp \exp \exp O(k))\), see [16]). For planar graphs, there exists a doubly exponential \((\exp \exp O(k))\) poly\((n)\) time algorithm, by Adler et al. [1]. For comparison, our running time dependence is singly exponential in the size of the terminal set, but of course our criteria for allowed connections is much relaxed.

The special case \( B = \emptyset \) in Disjoint \( A, B \)-paths is referred to as \( A \)-Paths in Lovász and Plummer [22]. Its solution in general undirected graphs by a polynomial time algorithm was given by Gallai [9] by a reduction to finding a perfect matching. Using Mulmuley, Vazirani, and Vazirani’s algorithm for the problem they call Exact Matching [23] on Gallai’s construction, one can in randomized polynomial time solve the Shortest disjoint \( A \)-paths.

The idea of using perfect matching counting in restricted graph classes to solve other combinatorial optimisation problems is not new, a classic example is the polynomial time algorithm for Max Cut in graphs of bounded genus by Galluccio, Lovász, and Vondrák [11].
An orientation of a graph $G$ is Pfaffian if and only if every even-length cycle $C$ such that $G \setminus V(C)$ has a perfect matching, has an odd number of edges directed in either direction along $C$. Kasteleyn [15], famously proved that all planar graphs have a Pfaffian orientation, and moreover showed how you given a planar graph can find a Pfaffian orientation fast. Nowadays it is even known how to find one in planar graphs in linear time, and Vazirani [30] showed it can be computed in NC. In general it only holds that $|\text{pm}(G)| = \left|\sqrt{\det(A_G)}\right|$, but we only consider positive edge weights in this paper and hence already know $\text{pm}(G)$ to be non-negative. Little [21] extended Kasteleyn’s method to also work constructively for graphs that do not have a $K_{3,3}$ subgraph as a minor. However, cubic $K_{3,3}$ minor free graphs coincide with the set of cubic planar graphs.

2.3 Reduction from Disjoint A,B-Paths to Counting Perfect Matchings

Consider as input a cubic planar graph $G$ and let $\ell: E \to \{1, \ldots, L\}$ be an edge length function, along with two disjoint subsets $A$ and $B$ of the vertices, each having even size. Set $A = \sum_{e \in E} \ell(e)$. We will reduce Shortest disjoint $A, B$-paths to counting perfect matchings so that planarity is preserved. In this section, we write $Z$ for the set of terminals, $Z = A \cup B$.

We build a larger graph $H$ from $G$ as follows. Replace each nonterminal $v \in V \setminus Z$ with three vertices $h_1(v)$, $h_2(v)$, and $h_3(v)$ forming a triangle. Replace each terminal $z \in Z$ by a 3-star on vertices $h_1(z)$, $h_2(z)$, $h_3(z)$, and terminal center $h(z)$. The gadgets look like this:

$$
\begin{array}{ccc}
& h_1(z) & \\
& h_2(z) & \Downarrow \\
h_3(v) & & h_3(z)
\end{array}
$$

We call the edges within these two gadgets internal edges.

Moreover, if $uv \in E(G)$, then $h_i(u)h_j(v)$ is also an edge in $H$ for some $i, j \in \{1, 2, 3\}$ in such a way that each vertex in $H$ is used in exactly one of the additional edges. We call these edges in $H$ between gadgets external edges. We write $f(uv) = h_i(u)$ and $g(uv) = h_j(v)$ to identify the two gadget vertices in $H$ connected by the external edge representing $uv$. Confer figure 2. The graph $H$ has the property that every vertex except the terminal centers $h(z)$ for $z \in Z$ is part of exactly one external edge. Our first insight is the following:

**Lemma 4.** If $G$ is planar, then so is $H$.

**Proof.** Both gadgets are easily seen to be planar. To see that $H$ is planar, use an embedding of $G$. For each vertex $v$ in $G$, consider a small enough circle $C_v$ around $v$ containing no other edge or vertex. Now replace $v$ with a copy of its gadget small enough to fit $C_v$. □

Hence, if $G$ is planar, we can find a Pfaffian orientation of $H$, as well as for any subgraph of it, as any subgraph is also planar. (We note in passing that a Pfaffian orientation of a graph is not necessarily a Pfaffian orientation of its subgraph.)

From $H$, we create several graphs depending on a subset of the terminal vertices. We write $H(X)$ for $X \subseteq Z$ to mean the graph obtained from $H$ by removing the terminal centers $h(z)$ and all incident edges for each terminal $z \not\in X$. We have $H = H(Z)$.

We introduce an indeterminate $s$ to control the length of the paths. We write $D(X, s)$ for a skew-symmetric adjacency matrix of a Pfaffian orientation of $H(X)$, where we have multiplied all entries representing an external edge $e$ in $H(X)$ with $s^{\ell(e)}$.

Our algorithm is a direct application of the following result:
Counting Shortest Two Disjoint Paths in Cubic Planar Graphs

Figure 2 Left: The instance graph $G$ with two terminal nodes $z_1$ and $z_2$. Middle: The gadget graph $H$. Right: The union of two matchings in $H(X)$ and $H(Z \setminus X)$ for some $X$ containing both $z_1$ and $z_2$. Together, they form a path between $h(z_1)$ and $h(z_2)$, along with three double edges.

Lemma 5. For a graph $G$, consider
\[
p(G, s) = \sum_{X \subseteq Z} (-1)^{|X \cap A|} \left| \sqrt{\det(D(X, s)) \det(D(Z \setminus X, s))} \right|
\]
as a polynomial in the indeterminate $s$. Let $cs^d$ be the largest degree monomial with a positive coefficient in $p(G, s)$. Then, $\ell_{A,B} = 2\Lambda - d$ is the shortest total length of any disjoint $A,B$-paths in $G$, and $c$ is $2^{|Z|/2}$ times the number of solutions having that minimum length.

Proof. We begin by arguing that $p(G, s)$ indeed is a polynomial in the indeterminate $s$. Fix $X \subseteq Z$. Write $M(X)$ for the set $M(H(X))$ of perfect matchings in $H(X)$. By (1), we have
\[
\left| \sqrt{\det(D(X, s))} \right| \leq \operatorname{pm}(H(X)) = \sum_{M \in M(X)} \prod_{e \in M} w(e),
\]
where $w(e) = s^{\ell(e)}$ if $e$ is external and $w(e) = 1$ if $e$ is internal. Thus, for a pair of perfect matchings $M_1 \in M(X)$ and $M_2 \in M(Z \setminus X)$, their contributing term $t(M_1, M_2)$ is
\[
t(M_1, M_2) = \left( \prod_{e \in M_1} w(e) \right) \cdot \prod_{e \in M_2} w(e),
\]
which is clearly a polynomial in $s$, and write
\[
p(G, s) = \sum_{X \subseteq Z} (-1)^{|X \cap A|} \sum_{M_1 \in M(X)} \sum_{M_2 \in M(Z \setminus X)} t(M_1, M_2).
\]

Now view $M_1 \cup M_2$ as a subgraph in $H$, by identifying each vertex in $H(X)$ and $H(Z \setminus X)$ with its copy in $H$. (It is helpful to view $M_1 \cup M_2$ as a multiset, so the corresponding subgraph is in fact a multigraph using the edges $M_1 \cap M_2$ twice.) We can visualise this as placing the two graphs on top of each other and looking at the subgraph formed by the two matchings. It is clear that every vertex in $H$ has degree at most 2 in this subgraph, so $M_1 \cup M_2$ can be partitioned into three edge subsets $\mathcal{P}, \mathcal{E}, \mathcal{D} \subseteq E(H)$, such that $\mathcal{P}$ is a disjoint union of simple paths, $\mathcal{E}$ is a disjoint union of simple cycles, and $\mathcal{D}$, which is equal to the intersection $M_1 \cap M_2$, is a disjoint union of isolated edges.

We claim that every path in $\mathcal{P}$ has its endpoints in terminal centers. To see this, first note that each terminal centre $h(z)$ for $z \in Z$ is present in exactly one of the graphs $H(X)$ and $H(Z \setminus X)$. Therefore, $h(z)$ is matched by exactly one edge in $M_1 \cup M_2$ and therefore is the endpoint of a path. Every other vertex in $H$ appears in both $H(X)$ and $H(Z \setminus X)$ and is therefore matched in both $M_1$ and $M_2$; in particular, no such vertex is the endpoint of a simple path. Figure 2 shows a small example.
We next argue that unions $M_1 \cup M_2$ whose paths connect terminal centers $h(a)$ and $h(b)$ with $a \in A$ and $b \in B$ contribute nothing to $p(G, s)$. To this end, consider such a term $t(M_1, M_2)$ with $M_1 \in M(X)$ and $M_2 \in M(Z \setminus X)$ and let $P = (u_1, \ldots, u_k)$ with $u_1 = h(a)$, $u_k = h(b)$ be the lexicographically first such path in $\mathcal{P}$.

If $k$ is odd, then the edges $u_1u_2, u_3u_4, \ldots, u_{k-2}u_{k-1}$ belong to one matching, say $M_1$, and the edges $u_2u_3, \ldots, u_{k-1}u_k$ belong to $M_2$. In particular, the terminal center $h(a)$ is matched in $M_1$, which implies $h(a) \in V(H(X))$ and therefore $a \in X$. Conversely, $h(b)$ is matched in $M_2$, which implies $h(b) \in V(H(Z \setminus X))$ and $b \notin X$. Now form $X' = (X \cup \{b\}) \setminus \{a\}$ and consider the two matchings $M'_1 \in M(X')$ and $M'_2 \in M(Z \setminus X')$ created from $M_1$ and $M_2$ by swapping the edges on $P$. Note that the edge $u_{k-1}u_k$ incident on $h(b)$ now belongs to $M'_1$, and since $b$ belongs to $X'$, the matching $M'_1$ is indeed a perfect matching in $M(X')$. Similarly, $M'_2 \in M(Z \setminus X')$. Starting the exact same process from the matchings $M'_1$ and $M'_2$ and set $X'$ would get us back to $M_1$, $M_2$, and $X$, since the same path $P$ will be chosen by the lexicographical order, so the process defines a fixed-point free involution on the set of terms $t(M_1, M_2)$ and subsets of $Z$. Crucially, the contribution to (2) of terms paired by this involution cancel:

$$(-1)^{|X' \cap A|}t(M_1, M_2) + (-1)^{|X' \cap A|}t(M'_1, M'_2) = 0,$$

because the multisets $M_1 \cup M_2$ and $M'_1 \cup M'_2$ are the same, and $X'$ and $X$ differ in exactly one terminal from $A$. Hence no such terms survive in the computation of $p(G, s)$.

If $k$ is even, then $u_1u_2, u_3u_4, \ldots, u_{k-2}u_{k-1}$ belong to the same matching, say $M_1$. Thus, both $h(a)$ and $h(b)$ belong to $H(X)$, so $a$ and $b$ belong to $X$. Set $X' = X \setminus \{a, b\}$, and follow the same argument as above.

In other words, $t(M_1, M_2)$ survives in $p(G, s)$ only if the disjoint paths in $\mathcal{P}$ have their endpoints either both in $A$ or both in $B$. The contribution is

$$t(M_1, M_2) = \left(\prod_{e \in D} w(e)^2\right) \cdot \prod_{e \in \mathcal{E} \cup \mathcal{P}} w(e) = s^d,$$

where

$$d = \left(2 \sum_{e \in D} \ell(e) + \sum_{e \in \mathcal{E} \cup \mathcal{P}} \ell(e)\right) = 2\Lambda - \sum_{e \in \mathcal{E} \cup \mathcal{P}} \ell(e),$$

with the convention that $\ell(e) = 0$ for internal edges. The last term is at least $\ell_{A,B}$, and attains that value exactly if $\mathcal{C}$ is empty and $\mathcal{P}$ contains the external edges of a solution $E'$ to Shortest disjoint $A, B$-paths in $G$. Otherwise, $d < 2\Lambda - \ell_{A,B}$.

We finally turn to the other direction, to show that if there exists disjoint $A, B$-paths in $G$, we will detect them in $p(G, s)$. Moreover, we argue that we can count the ones of shortest total length. To see this, first consider a solution $E' \subseteq E(G)$ to Shortest disjoint $A, B$-paths, i.e., a disjoint union of paths

$$E' = P_1 \cup \cdots \cup P_{|Z|/2},$$

each of which has terminal endpoints either both in $A$, or both in $B$. Let $T$ be a subgraph of $H$ obtained in the following way. For each such path $P = (v_1, \ldots, v_k)$, first add the external edges $f(v_1v_2)g(v_1v_2), \ldots, f(v_{k-1}v_k)g(v_{k-1}v_k)$ to $T$. Second, add the internal edges $h(v_1)f(v_1v_2)$ and $g(v_{k-1}v_k)h(v_k)$ in the two terminal gadgets, and the internal edges $g(v_iv_{i+1})f(v_{i+1}v_{i+2})$ in the nonterminal gadgets for $i \in \{1, \ldots, k - 2\}$. This adds precisely one internal edge per gadget representing a vertex on $P$. Third, for every vertex $u \in V(H)$
not used in an edge so far, we add to $T$ its unique external edge in $H$. This is where we use the property of $H$ that every non-terminal vertex has a unique external edge. Thus, $T$ consists of disjoint edge sets $\mathcal{P}, \mathcal{D} \subseteq E(H)$ where $\mathcal{P}$ consists of disjoint paths and $\mathcal{D}$ consists of disjoint (external) edges.

We continue to account for the contribution of $T$ to (2). Let $X \subseteq Z$ be a subset of terminals such that the endpoints of the paths in $\mathcal{P}$ are either both in $X$ or both in $Z \setminus X$. In particular, $|X \cap A|$ is even, and there are $2^{|X|/2}$ such subsets. There is exactly one perfect matching $M_1$ in $H(X)$ that is a subgraph of $T$; this matching contains all the internal edges on the paths of $\mathcal{P}$ with endpoints both in $X$. There is also exactly one perfect matching $M_2$ in $H(Z \setminus X)$ that is a subgraph of $T$; this matching contains all the external edges on the paths of $\mathcal{P}$ with endpoints both in $Z \setminus X$. In particular, every external edge in $\mathcal{D}$ appears exactly twice in the multiset $M_1 \cup M_2$, and every external edges in $\mathcal{P}$ appears exactly once. (The internal edges have weight 1, so we need not count their contribution to a product.)

Thus, the total contribution of $M_1$ and $M_2$ is

$$t(M_1, M_2) = \left( \prod_{e \in \mathcal{D}} w(e)^2 \right) \cdot \prod_{e \in \mathcal{P}} w(e) = s^d,$$

where $d = 2\Lambda - \ell_{A,B}$,

and the solution $E'$ accounts for the contribution

$$\sum_{X \subseteq Z} (-1)^{|X \cap A|} t(M_1, M_2) = 2^{|Z|/2} s^d.$$

All other surviving terms have lower degree in $p(G, s)$, and the lemma follows. \hfill \blacktriangle

### 2.4 Algorithm

Our algorithm computes the coefficients of $p(G, s)$ in (2) viewed as a polynomial in $s$, using polynomial interpolation. The algorithm works through direct evaluation in sufficiently many points $s \in \{0, 1, \ldots, 2\Lambda\}$ of $p(G, s)$ after replacing $s$ for its numerical value. Hence all computations are over the integers.

1. For $s = 0$ to $2\Lambda$.
2. Set $\text{sum}_s = 0$.
3. For $X \subseteq Z$, $|X|$ even,
   4. Construct $H(X)$ and $H(Z \setminus X)$ and their Pfaffian orientations.
   5. Compute the integers $\det^2(D(X))$ and $\det^2(D(Z \setminus X))$ for the current value of $s$.
   6. Take the fourth root of the two determinants and multiply them.
   7. Add the product with the sign $(-1)^{|X \cap A|}$ to $\text{sum}_s$.
8. Use polynomial interpolation to compute the coefficients of $p(G, s)$ from the array $\text{sum}$.
9. Locate the largest non-zero monomial $cs^d$.
10. Return $\ell_{A,B} = 2\Lambda - d$ and $S_{A,B} = c/2^{|Z|/2}$.

### 2.5 Sequential Running Time

We turn to the sequential running time of the algorithm from section 2.4. Recall that the polynomial $p(G, s)$ has degree at most $2\Lambda$, and hence the number of evaluated points is sufficient to uniquely recover its coefficients. We can bound the value of the two determinants using the Leibniz formula for the determinant. There are at most $3^{3n+|A\cup B|}$ terms since there are at most 3 choices per vertex in $H(X)$. For each choice, the largest value is obtained when the external edges are picked twice, i.e., every term is at most $(2\Lambda)^{2\Lambda}$. Hence the
determinant can be a $\beta = \tilde{O}(nL)$ bit number. We can compute the determinants in row 5 using $O(n^{\omega/2})$ arithmetic operations, using Yuster’s algorithm [32] for the square of the determinant, which in turn uses the dissection method developed by Lipton et al. [20]. Note that since we know all our determinants to be positive (each is the square of the number of perfect matchings), no information is lost by computing even powers of the determinant. Every arithmetic operation can be computed in $\tilde{O}(\beta)$ time [31]. Computing all determinants requires at most $O(\Lambda^2|\mathcal{A}\cup\mathcal{B}|n^{\omega/2}\beta) = \tilde{O}(2^{\mathcal{A}\cup\mathcal{B}}n^{\omega/2+2L^2})$ time. This part dominates the computation time, since taking the square roots in row 7 using Newton’s method requires only about $\log nL$ iterations for an integer square, and the polynomial interpolation in row 8 can be done in quadratic time. It requires $\tilde{O}(\Lambda)$ operations over a finite field, cf. [31], and we need a field, or several fields and the Chinese remainder theorem, of total size $\Omega(\beta)$ to recover the integer values. This completes the proof of Theorem 1.

2.6 Parallel Circuit Depth

In this section we prove that our algorithm in section 2.4 can be efficiently implemented as a circuit of polynomial size and polylogarithmic depth. First we note that all values of $s$ and all values of $X$ in row 1 and 3 of the algorithm can be evaluated in parallel. All computations are made on integers of $\beta = \tilde{O}(nL)$ bits as claimed in the previous section. Addition and multiplication on $\beta$ bit integers can be done in polylog($\beta$) depth. Constructing the graphs $H(X)$ in row 4 can be done even without a planar embedding of $G$, as it doesn’t matter how the external edges are mapped to the gadget’s connectors, planarity is always preserved. Vazirani shows that the number of perfect matchings can be computed by an NC algorithm [30], see also the textbook [14]. He describes how a Pfaffian orientation for a planar graph can be obtained via Klein and Reif’s parallel planar embedding algorithm [18]. He next uses the fact that the determinant can be computed in NC, a consequence of Csanky’s algorithm for the determinant [6]. Berkowitz algorithm [3] via iterated matrix product can also be used (see Cook [5]). Computing the integer square root at row 6 is a logarithmic depth task with Newton’s method since the convergence is quadratic. Once all evaluations are done, the inner loop summation at row 7 can be computed for all $s$, again in polylogarithmic depth by a balanced binary tree of adders of $\beta$-sized integers. Finally, Cook describes how polynomial interpolation is in NC [5] by reducing to Berkowitz algorithm for the determinant [3]. This completes the proof of Theorem 2.

3 Hardness Result: Theorem 3

Our hardness reduction is from counting maximum independent sets in cubic planar graphs, proven #P-hard in Vadhan [29] (Corollary 4.2.1). The NP-hardness result for Disjoint $A,B$-paths in general graphs by Hirai and Namba [12], follows Hirai and Pap [13]. It is a reduction directly from 3-Satisfiability but it is not (weakly) parsimonious. We give here such a strengthened reduction.

Consider a cubic planar graph $G$ in which we want to count the maximum independent sets. From $G$, we construct a maximum degree 3 planar instance $I$ to Shortest disjoint $A,B$-paths. As described in the previous section, we can by adding a few vertices per vertex of degree less than three make sure the graph is cubic while preserving planarity. Here we will stick with a few vertices of degree two in our description of $I$ for simplicity. First, for every vertex $v \in V$, we add a clockwise ordered cycle $v_1', \ldots, v_8'$. The edge $v'_8v'_1$ has length 12 whereas all other edges $v'_i v'_{i+1}$ for $i \in \{1, \ldots, 7\}$ have length 2 if $i$ is odd and length 1 if $i$ is even. Furthermore, vertices $v'_1$ and $v'_8$ belong to $A$ for every vertex $v$. Second, for every edge $uv \in E$, we add two vertices $w'_1$ and $w'_2$ to $I$. We add edges $w'_1 u'_i, w'_2 u_{i+1}, w'_1 v'_j$, and
w'_i v_j of length 1 for some indices i and j so that no vertex is used more than once, and the resulting graph I is planar. This is easy to accomplish by using a planar embedding of G and order edges incident on a vertex in clockwise order. See Figure 3. Furthermore, we add w'_1 and w'_2 to B.

Lemma 6. Let \( \ell_{A,B} \) and \( S_{A,B} \) be the solution to Shortest disjoint \( A,B \)-paths on I, then the maximum independent set in G has size \( \alpha(G) = 12|V| + 3|E(G)| - \ell_{A,B} \) and the number of such sets are \( S_{A,B}/2^{\lfloor E(G)\rfloor - 3\alpha(G)} \).

Proof. Any vertex pair in A on the same vertex gadget cycle must be connected with each other through a path, since there are no paths between different vertex gadget cycles that do not also pass through a terminal in B. Hence there are only two possibilities for every such pair: either it is connected through the 12-long edge between them, or it uses the path around the cycle of length 11. Let I \( \subseteq V \) be the set of vertices whose vertex gadgets uses paths of length 11 to connect its two A terminals. The set I must be an independent set in G, since the terminals on every edge gadget must use some edge on either of the two vertex gadgets it is connected to. Moreover, any pair of terminals in B cannot be connected with a path shorter than 3 as there exist no such short paths between any pair of them. A lower bound on the attainable length of a Shortest disjoint \( A,B \)-paths solution is hence \( 12|V| - \alpha(G) + 3|E| \), where \( \alpha(G) \) is the size of a maximum independent set in G. Any such solution can naturally be interpreted as a maximum independent set in G by identifying the A-paths of length 11.

Moreover, from any maximum independent set I in G, we can construct disjoint paths of this length, simply by taking the 12-long edge for every vertex not in I for a vertex gadget’s A terminals, and the shorter 11-long path for the other vertex gadgets. The edge gadgets’ B terminals can be connected pairwise with each other through a 3-long path using an edge on either of its two adjacent vertex gadgets, whenever it represents a vertex not in I. It might be possible to connect the B terminals in other ways, but those paths will be of length strictly longer than 3 as they need to use a 2-long edge on some vertex gadget. There are precisely \( |E| - 3\alpha(G) \) edges with neither endpoint in I, and hence the maximum independent sets will be counted \( 2^{\lfloor E\rfloor - 3\alpha(G)} \) times in the Shortest disjoint \( A,B \)-paths.

Theorem 3 now directly follows from Lemma 6, since if we can find \( \ell_{A,B} \) in I, we can also compute the number of maximum independent sets in G from \( S_{A,B} \).

References


A Appendix

We extend the algorithm to planar graphs of maximum degree 3. First consider an edge $ua$ where $u$ is a nonterminal vertex of degree 3 and $a$ is a terminal of degree 1. Then $ua$ can be removed and $u$ inserted into the terminal set of $a$; the resulting instance has a shortest solution of size $\ell_{A,B} - \ell(ua)$. When $u$ is also a terminal vertex, there are two cases: If $u$ belongs to the same terminal set as $a$ then $ua$ must be a path in the shortest solution, so we can remove both $u$ and $a$ and discount the resulting value by $\ell(ua)$. If $u$ belongs to the other terminal set than $a$ then there is no solution and we can output $S_{A,B} = 0$.

Consider a $u,v$-path $P$ whose internal vertices all have degree 2. If none of $P$’s internal vertices are terminals then $P$ can be contracted into a single edge with the sum of the original edge lengths. If $P$ contains alternating terminals, say $a \in A$, $b \in B$, $a' \in A$ in that order, no solution can exist. If $P$ contains exactly two terminals $a \in A$ and $b \in B$ then its prefix from $u$ to $a$ can be contracted into a single edge, and so can its suffix from $b$ to $v$; the infix from $a$ to $b$ can be removed. The resulting dangling edges $ua$ and $vb$ are handled as above.

In general, we can replace a degree-2 terminal $a$ incident on the edges $ua$ and $av$ with the 4-vertex ‘diamond’ graph, introducing 3 new nonterminal vertices. The original edges retain their lengths, and the new edges receive length 1, so that

\[
\begin{array}{c}
\ell_1 \\
\hline \\
\ell_2 \\
\end{array}
\quad \rightarrow \quad
\begin{array}{c}
\ell_1 \\
\hline \\
\ell_2 \\
\end{array}
\quad \rightarrow \quad
\begin{array}{c}
\ell_1 \\
\hline \\
\ell_2 \\
\end{array}
\]
No path with endpoint $a$ in an optimal solution uses $w$, because $uu'a$ is shorter than $uu'wa$. No other solution uses the nonterminal $w$ either, because this would isolate $a$. Thus, an optimal solution uses either $au'u$ or $av'v$ and no other edges in the gadget. We conclude that every optimal solution in the transformed graph corresponds to exactly one optimal solution in the original, and $\ell_{A,B}$ increments by one for each of these modifications.
Data-Compression for Parametrized Counting Problems on Sparse Graphs

Eun Jung Kim
Université Paris-Dauphine, PSL Research University, CNRS/LAMSADE, 75016, Paris, France
eun-jung.kim@dauphine.fr

Maria Serna
Computer Science Department & BGSMath, Universitat Politècnica de Catalunya, Barcelona, Spain
mjserna@cs.upc.edu

Dimitrios M. Thilikos
AlGCo project-team, LIRMM, Université de Montpellier, CNRS, Montpellier, France; and Department of Mathematics, National and Kapodistrian University of Athens, Greece
sedthilk@thilikos.info

Abstract

We study the concept of compactor, which may be seen as a counting-analogue of kernelization in counting parameterized complexity. For a function $F : \Sigma^* \rightarrow \mathbb{N}$ and a parameterization $\kappa : \Sigma^* \rightarrow \mathbb{N}$, a compactor $(P, M)$ consists of a polynomial-time computable function $P$, called condenser, and a computable function $M$, called extractor, such that $F = M \circ P$, and the condensing $P(x)$ of $x$ has length at most $s(\kappa(x))$, for any input $x \in \Sigma^*$. If $s$ is a polynomial function, then the compactor is said to be of polynomial-size. Although the study on counting-analogue of kernelization is not unprecedented, it has received little attention so far. We study a family of vertex-certified counting problems on graphs that are MSOL-expressible; that is, for an MSOL-formula $\phi$ with one free set variable to be interpreted as a vertex subset, we want to count all $A \subseteq V(G)$ where $|A| = k$ and $(G, A) \models \phi$. In this paper, we prove that every vertex-certified counting problems on graphs that is MSOL-expressible and treewidth modulable, when parameterized by $k$, admits a polynomial-size compactor on $H$-topological-minor-free graphs with condensing time $O(k^2n^2)$ and decoding time $2^{O(k)}$. This implies the existence of an FPT-algorithm of running time $O(n^2k^2 + 2^{O(k)})$. All aforementioned complexities are under the Uniform Cost Measure (UCM) model where numbers can be stored in constant space and arithmetic operations can be done in constant time.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases Parameterized counting, compactor, protrusion decomposition

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.20

Related Version The full version of this extended abstract has appeared in [35], http://arxiv.org/abs/1809.08160.
1 Introduction

A large part of research on parameterized algorithms has been focused on algorithmic techniques for parametrizations of decision problems. However, relatively less effort has been invested for solving parameterized counting problems. In this paper, we provide a general data-reduction concept for counting problems, leading to a formal definition of the notion of a compactor. Our main result is an algorithmic meta-theorem for the existence of a polynomial size compactor, that is applicable to a wide family of problems of graphs.

1.1 General context

**Algorithmic meta-theorems.** Parameterized complexity has been proposed as a multi-variable framework for coping with the inherent complexity of computational problems. Nowadays, it is a mature discipline of modern Theoretical Computer Science and has offered a wealth of algorithmic techniques and solutions (see [12, 17, 22, 38] for related textbooks). In some cases, in-depth investigations on the common characteristics of parameterized problems gave rise to algorithmic meta-theorems. Such theorems typically provide conditions, logical and/or combinatorial, for a problem to admit a parameterized algorithm [30, 29, 36, 40]. Important algorithmic meta-theorems concern model-checking for Monadic Second Order Logic (MSOL) [10, 7, 2, 41] on bounded treewidth graphs and model checking for First Order Logic (FOL) on certain classes of sparse graphs [21, 28, 13, 20, 19, 31].

In some cases, such theorems have a counterpart on counting parameterized problems. Here the target is to prove that counting how many solutions exist for a problem is fixed-parameter tractable, under some parameterization of it. Related meta-algorithmic results concern counting analogues of Courcelle’s theorem, proved in [9], stating that counting problems definable in MSOL are fixed-parameter tractable when parameterized by the tree-width of the input graph. Also similar results for certain fragments of MSOL hold when parameterized by the rank-width of the input graph [9]. Moreover, it was shown in [27] that counting problems definable in first-order logic are fixed-parameter tractable on locally tree-decomposable graphs (e.g. for planar graphs and bounded genus graphs).

**Kernelization and data-reduction.** A well-studied concept in parameterized complexity is kernelization. We say that a parameterized problem admits a polynomial kernel if there is an algorithm – called kernelization algorithm – that can transform, in polynomial time, every input instance of the problem to an equivalent one, whose size is bounded by a function of the parameter. When this function is polynomial then we have a polynomial kernel. A polynomial kernel permits the drastic data-reduction of the problem instances to equivalent “miniatures” whose size is independent from the bulk of the input size and is polynomial on the parameter. That way, a polynomial kernel, provides a preprocessing of computationally hard problems that enables the application of exact algorithmic approaches (however still super-polynomial) on significantly reduced instances [37].

**Meta-algorithmic results for kernelization.** Apart from the numerous advances on the design of polynomial kernels for particular problems, algorithmic meta-theorems appeared also for kernelization. The first result of this type appeared in [4], where it was proved that certain families of problems on graphs admit polynomial kernels on bounded genus graphs. The logic-condition of [4] is CMSOL-expressibility or, additionally, the Finite Integer Index (FII) property (see [1, 6, 14]). Moreover, the meta-algorithmic results of [4] require additional combinatorial properties for the problems in question. The results in [4] where
extended in [23] (see also [25]) where the combinatorial condition for the problem was related
to bidimensionality, while the applicability of the results was extended in minor-closed graph
classes. Finally, further extensions appeared in [34] where, under the bounded treewidth-
modulability property (see Subsection 1.2), some of the results in [23, 4] could be applied to
more graph classes, in particular those excluding some fixed graph as a topological minor.

Data reduction for counting problems. Unfortunately, not much has been done so far in
the direction of data-reduction for parameterized counting problems. The most comprehen-
sive work in this direction was done by Marc Thurley [42] (see also [43]) who proposed the
first formal definition of a kernelization analogue for parameterized problems called counting
kernelization. In [42] Thurley investigated up to which extent classic kernelization techniques
such as Buss’ Kernelization and crown decomposition may lead to counting counterparts
of kernelization. In this direction, he provided counting kernelizations for a series of para-
terized counting problems such as and p-#VertexCover, p-card-#Hitting Set and p-#Unique Hitting Set.

Compactor enumeration. Another framework for data-reduction on parameterized counting
problems is provided by the notion of a compactor. In a precursory level, it appeared for
the first time in [16]. The rough idea in [16] was to transform the input of a parameterized
counting problem to a structure, called the compactor, whose size is (polynomially) bounded
by the parameter and such that the enumeration of certain family of objects (referred as
compactor enumeration in [16]) in the compactor is able to derive the number of solutions
for the initial instance. This technique was introduced in [16] for counting restrictive list
H-colorings and, later in [39], for counting generalized coverings and matchings. However
none of [16, 39] provided a general formal definition of a compactor, while, in our opinion,
the work of Thurley provides a legitimate formalization of compactor enumeration.

In this paper, we define formally the concept of a compactor for parameterizations of
function problems (that naturally include counting problems) that is not based on enumeration.
As a first step, we observe that for parameterized function problems, the existence of a
compactor is equivalent to the existence of an FPT-algorithm, a fact that is also the case for
classic kernels on decision problems and for counting kernels in [42].

Under the above formal framework, we prove an algorithmic meta-theorem on the existence
of polynomial compactors for a general family of graph problems. In the next subsection, we
define the compactor concept and we present the related meta-algorithmic results.

1.2 Our results

Counting problems and parameterizations. First of all notice that, for a counting problem,
it is not possible to have a kernelization in the classic sense, that is to produce an reduced
instance, bounded by a function of $k$, that is counting-equivalent in the sense that the number
of solutions in the reduced instance will provide the number of solutions in the original
one. For this reason we need a more refined notion of data compression where we transform
the input instance to “structure”, whose size is bounded by a function of $k$. This structure
contains enough information (combinatorial and arithmetical) so as to permit the recovering
of the number of the solutions in the initial instance. We next formalize this idea to the
concept of a compactor.

Let $\mathbb{N}$ be all non-negative integers and by $\text{poly}$ the set of all polynomials. Let $\Sigma$ be a
fixed alphabet. A parameterized function problem is a pair $(F, \kappa)$ where $F, \kappa : \Sigma^* \to \mathbb{N}$. An FPT-algorithm for $(F, \kappa)$ is one that, given $x \in \Sigma^*$, outputs $F(x)$ in $f(\kappa(x)) \cdot \text{poly}(|x|)$
steps. When evaluating the running time, we use the standard Uniform Cost Measure (UCM) model where all basic arithmetic computations are carried out in constant time. We also disregard the size of the numbers that are produced during the execution of the algorithm.

**Compactors.** Let \((F, \kappa)\) be a parameterized function problem. A *compactor* for \((F, \kappa)\) is a pair \((P, M)\) where

- \(P : \Sigma^* \to \Sigma^*\) is a polynomially computable function, called an *condenser*,
- \(M : \Sigma^* \to \mathbb{N}\) is a computable function, called a *extractor*,
- \(F = M \circ P\), i.e., \(\forall x \in \Sigma^*\), \(F(x) = (M \circ P)(x)\), and
- there is a recursive function \(s : \mathbb{N} \to \mathbb{N}\) where \(\forall x \in \Sigma^*\), \(|P(x)| \leq s(\kappa(x))\).

We call the function \(s\) size of the compactor \((P, M)\) and, if \(s \in \text{poly}\), we say that \((P, M)\) is a *polynomial-size compactor* for \((F, \kappa)\). We call the running time of the algorithm computing \(P\), measured as a function of \(|x|\), *condensing time of \((P, M)\)*. We also call the running time of the algorithm computing \(M\), measured as a function of \(\kappa(x)\), *decoding time of \((P, M)\)*. We can readily observe that parameterized function problem has an FPT-algorithm if and only if there is a compactor for it.

Up to our best knowledge, the notion of compactor as formalized in this paper is new. As discussed in Subsection 1.1, similar notions have been proposed such as counting kernelization [42] and compactor enumeration [16]. In both counting kernelization and compact enumeration, a mapping from the set of all certificates to certain objects in the new instance is required. While this approach comply more with the idea of classic kernelization, it seems to be more restrictive. The main difference of our compactor from the previous notions is that (the condenser of) a compactor is free of this requirement, which makes the definition more flexible and easier to work with. Due to this flexibility and succinctness, we believe that our notion might be amenable for lower bound machineries akin to those for decision problem kernelizations.

**Parameterized counting problems on graphs.** A *structure* is a pair \((G, A)\) where \(G\) is a graph and \(A \subseteq V(G)\). Given a MSOL-formula \(\phi\) on structures and some graph class \(\mathcal{G}\), we consider the following parameterized counting problem \(\Pi_{\phi, \mathcal{G}}\):

<table>
<thead>
<tr>
<th>(\Pi_{\phi, \mathcal{G}})</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> a graph (G \in \mathcal{G}), an non-negative integer (k).</td>
</tr>
<tr>
<td><strong>Parameter:</strong> (k).</td>
</tr>
<tr>
<td><strong>Count:</strong> the number of vertex sets (A \subseteq V(G)) such that ((G, A) \models \phi) and (</td>
</tr>
</tbody>
</table>

We say that an instance \((G, k) \in \mathcal{G} \times \mathbb{N}\) of \(\Pi_{\phi, \mathcal{G}}\) is a *null* instance if it has no solutions. Given a graph \(G\), we say that a vertex set \(A \subseteq V(G)\) is a *t treewidth modulator* of \(G\) if the removal of \(A\) from \(G\) leaves a graph of treewidth at most \(t\). Given an MSOL-formula \(\phi\) and a graph class \(\mathcal{G}\), we say that \(\Pi_{\phi, \mathcal{G}}\) is *treewidth modulable* if there is a constant \(t\) (depending on \(\phi\) and \(\mathcal{G}\) only) such that, for every non-null instance \((G, k)\) of \(\Pi_{\phi, \mathcal{G}}\), \(G\) has a \(t\)-treewidth modulator of size at most \(t \cdot k\).

Let \(\mathcal{F}_H\) be the class of all graphs that do not contain a subdivision of \(H\) as a subgraph. The next theorem states our main result.

**Theorem 1.** For every graph \(H\) and every MSOL-formula \(\phi\), if \(\Pi_{\phi, \mathcal{F}_H} \) is treewidth modulable, then there is a compactor for \(\Pi_{\phi, \mathcal{F}_H}\) of size \(O(k^2)\) with condensing time \(O(k^2n^2)\) and decoding time \(2^{O(k)}\).

As a corollary of the main theorem we have the following.
Corollary 2. For every graph $H$ and every MSOL-formula $\phi$, if $\Pi_{\phi,H}$ is treewidth modulable, then $\Pi_{\phi,H}$ can be solved in $O(k^2n^2) + 2^{O(k)}$ steps.

In the above results, the constants hidden in the $O$-notation depend on the choice of $\phi$, on the treewidth-modulability constant $t$, and on the choice of $H$.

Recall that the above results are stated using the UCM model. As for $\Pi_{\phi,H}$, the number of solutions is $O(n^k)$ and this number can be encoded in $O(k \log n)$ bits. Assuming that summations of two $r$-bit numbers can be done in $O(r)$ steps and multiplications of two $r$-bit numbers can be done in $O(r^2)$ steps, then the size of the compactor in Theorem 1 is $O(k^2 \log n)$ the condensing and extracting times are $O(k^4 n^2 \log^2 n)$ and $2^{O(k)} \log^2 n$ respectively. Consequently, the running time of the algorithm in Corollary 2 is $O(k^4 n^2 \log^2 n) + 2^{O(k)} \log^2 n$.

Coming back to the algorithmic meta-theorems on parameterized counting problems we should remark that the problem condition of Corollary 2 is weaker than MSOL, as it additionally demands treewidth-modulability. However, the graph classes where this result applies have unbounded treewidth or rankwidth. That way our results can be seen as orthogonal to those of [9].

On the side of FOL, the problem condition of Corollary 2 is stronger than FOL, while its combinatorial applicability includes planar graphs or graphs of bounded genus where, the existing algorithmic meta-theorems require FOL-expressibility (see [27]).

1.3 Outline of the compactor algorithms

Our approach follows the idea of applying data-reduction based on protrusion decomposability. This idea was initiated in [4] for the automated derivation of polynomial kernels on decision problems. The key-concept in [4] is the notion of a protrusion, a set of vertices with small neighborhood to the rest of the graph and inducing a graph of small treewidth. Also, [4] introduced the notion of a protrusion decomposition, which is a partition of $G$ to $O(k)$ graphs such the first one is a “center”, of size $O(k)$, and the rest are protrusions whose neighborhoods are in the center.

The meta-algorithmic machinery of [4] is based on the following combinatorial fact: for the problems in question, $\text{YES}$-instances – in our case non-null instances– admit a protrusion decomposition that, when the input has size $\Omega(k)$, one of its protrusions is “big enough”. This permits the application of some “graph surgery” that consists in replacing a big protrusion with a smaller one and, that way, creates an equivalent instance of the problem (the replacements are based on the MSOL-expressibility of the problem). In the case of counting problems, this protrusion replacement machinery does not work (at least straightforwardly) as we have to keep track, not only of the way some part of a solution “invades” a protrusion, but also of the number of all those partial solutions. Instead, we take another way that avoids stepwise protrusion replacement. In our approach, the condenser of the compactor first constructs an approximate protrusion decomposition, then, it computes how many possible partial solutions of all possible sizes may exist in each one of the protrusions. This computation is done by dynamic programming (see Section 4) and produces a total set of $O(k^2)$ arithmetic values. These values, along with the combinatorial information of the center of the protrusion decomposition and the neighborhoods of the protrusions in the center, constitutes the output of the condenser. This structure can be stored in $O(k^2)$ space (given that arithmetic values can be stored in constant space) and contains enough information to obtain the number of all the solutions of the initial instance in $2^{O(k)}$ steps (Section 4).

We stress that the above machinery demands the polynomial-time construction of a constant-factor approximation of a protrusion-decomposition. To our knowledge, this remains an open problem in general. So far, no such algorithm has been proposed, even for particular
graph classes, mostly because meta-kernelization machinery in [4] (and later in [25, 23, 34, 24]) is based on stepwise protrusion replacement and does not actually need to construct such a decomposition. Based on the result in [34], we show that the construction of such an approximate protrusion decomposition is possible on $H$-topological-minor-free graphs, given that it is possible to construct an approximate $t$-treewidth modulator of $G$. In fact, this can been done in general graphs using the randomized constant-factor approximation algorithm in [24]. Responding to the need for a deterministic approximation we provide a constant-factor approximation algorithm that finds a $t$-treewidth modulator on $H$-topological-minor free graphs (Section 3). This algorithm runs in $O(k^2n^2)$ steps and, besides from being a necessary step of the condenser of our compactor, is of independent algorithmic interest.

2 Preliminaries

We use $\mathbb{N}$ to denote the set of all non-negative integers. Let $\chi : \mathbb{N}^2 \to \mathbb{N}$ and $\psi : \mathbb{N} \to \mathbb{N}$. We say that $\chi(n,k) = O_k(\psi(n))$ if there exists a function $\phi : \mathbb{N} \to \mathbb{N}$ such that $\chi(n,k) = O(\phi(k) \cdot \psi(n))$. Given $a, b \in \mathbb{N}$, we define $[a,b] = \{a, \ldots, b\}$. Also, given some $a \in \mathbb{N}$ we define $[a] = \{1, \ldots, a\}$. Given a set $Z$ and a $k \in \mathbb{N}$, we denote $\binom{k}{B} = \{Z \subseteq Z \mid |Z| = k\}$.

2.1 Graphs and boundary graphs

Graphs. All graphs in this paper are simple and undirected. Given a graph $G$, we use $V(G)$ to denote the set of its vertices. Given a $S \subseteq V(G)$ we denote by $N_G(S)$ the set of all neighbours of $S$ in $G$ that are not in $S$. We also set $N_G[S] = S \cup N_G(S)$ and we use $N(S)$ and $N[S]$ as shortcuts of $N_G(S)$ and $N_G[S]$ (when the index is a graph denoted by $G$). We define $G - S$ as the graph obtained from $G$ if we remove the vertices in $S$, along with the edges incident to them. The subgraph of $G$ induced by $S$ is the graph $G|S := G - (V(G) \setminus S)$. Finally, we set $\partial G(S) = N_G(V(G - S))$. We call $|V(G)|$ the size of a graph $G$ and $n$ is reserved to denote the size of the input graph for time complexity analysis.

Given a graph $G$, a subdivision of $G$ is any graph that is obtained from $G$ after replacing its edges by paths with the same endpoints. We say that a graph $H$ is a topological minor of $G$ if $G$ contains as a subgraph some subdivision of $H$. We also say that $G$ is $H$-topological-minor-free if it excludes $H$ as a topological minor.

Boundaried structures. A labeling of a graph $G$ is any injective function $\lambda : V(G) \to \mathbb{N}$. Given a structure $(G,A)$, we call $A$ the annotated set of $(G,A)$ and the vertices in $A$ annotated vertices of $(G,A)$.

A boundaried structure, in short a $b$-structure, is a triple $\mathbf{G} = (G,B,A)$ where $G$ is a graph and $B,A \subseteq V(G)$. We say that $B$ is the boundary of $\mathbf{G}$ and $A$ is the annotated set of $G$. Also we call the vertices of $B$ boundary vertices and the vertices in $A$ annotated vertices. We use notation $B^{(t)}$ to denote all $b$-structures whose boundary has at most $t$ vertices. We set $G(\mathbf{G}) = G$, $V(\mathbf{G}) = V(G)$, $B(\mathbf{G}) = B$, $A(\mathbf{G}) = A$. We refer to $G$ as the underlying graph of $\mathbf{G}$ and we always assume that the underlying graph of a $b$-structure is accompanied with some labelling $\lambda$. Under the presence of such a labelling, we define the index of a boundary vertex $v$ as the quantity $|\{u \in B \mid \lambda(u) \leq \lambda(v)\}|$ i.e., the index of $v$ when we arrange the vertices of $B$ according to $\lambda$ in increasing order. We extend the notion of index to subsets of $B$ in the natural way, i.e., the index of $S \subseteq B$ consists of the indices of all the vertices in $S$.

A boundaried graph, in short $b$-graph, is any $b$-structure $\mathbf{G} = (G,B,A)$ such that $A = V(G)$. For simplicity we use the notation $\mathbf{G} = (G,B,-)$ to denote $b$-graphs instead of using the heavier notation $\mathbf{G} = (G,B,V(G))$. For every $t \in \mathbb{N}$, we use $\mathbf{B}^{(t)}$ to denote the
b-graphs in $\mathcal{B}^{(t)}$. We avoid denoting a boundary graph as an annotated graph as we want to stress the role of $B$ as a boundary.

We say that two b-structures $G_1 = (G_1, B_1, A_1)$ and $G_2 = (G_2, B_2, A_2)$ are compatible, denoted by $G_1 \sim G_2$, if $A_1 \cap B_1$ and $A_2 \cap B_2$ have the same index and the labeled graphs $G[B_1]$ and $G[B_2]$, where each vertex of $B_i$ is labeled by its index, are identical.

Given two compatible b-structures $G_1 = (G_1, B_1, A_1)$ and $G_2 = (G_2, B_2, A_2)$, we define $G_1 \oplus G_2$ as the structure $(G, A)$ where

- the graph $G$ is obtained by taking the disjoint union of $G_1$ and $G_2$ and then identifying boundary vertices of $G_1$ and $G_2$ of the same index, and
- the vertex set $A$ is obtained from $A_1$ and $A_2$ after identifying equally-indexed vertices in $A_1 \cap B_1$ and $A_2 \cap B_2$.

Keep in mind that $(G, A) = G_1 \oplus G_2$ is an annotated graph and not a b-structure. We always assume that the labels of the boundary of $G_1$ prevail during the gluing operation, i.e., they are inherited to the identified vertices in $(G, A)$ while the labels of the boundary of $G_2$ disappear in $(G, A)$. Especially, when $G_1$ and $G_2$ are compatible b-graphs, we treat $G_1 \oplus G_2$ as a graph for notational simplicity.

**Treewidth of b-structures.** Given a b-structure $G = (G, B, A)$, we say that the triple $D = (T, \chi, r)$ is a tree decomposition of $G$ if $(T, \chi)$ is a tree decomposition of $G$, $r \in V(T)$, and $\chi(r) = B$. We see $T$ as a tree rooted on $r$. The width of a tree decomposition $D = (T, \chi, r)$ is the width of the tree decomposition $(T, \chi)$. The treewidth of a b-structure $G$ is the minimum width over all its tree decompositions and is denoted by $\tw(G)$. We use $T^{(t)}$ (resp. $\mathcal{T}^{(t)}$) to denote all b-structures (resp. b-graphs) in $\mathcal{B}^{(t)}$ (resp. $\mathcal{B}^{(t)}$) with treewidth at most $t$.

**Protrusion decompositions.** Let $G$ be a graph. Given $\alpha, \beta, \gamma \in \mathbb{N}$, an $(\alpha, \beta, \gamma)$-protrusion decomposition of $G$ is a sequence of $G_1 = (G_1, B_1, -), \ldots, G_s = (G_s, B_s, -)$ of b-graphs where, given that $X_i = V(G_i) \setminus B_i$, $i \in [s]$, it holds that

1. $s \leq \alpha$
2. $\forall i \in [s], \ G_i \in \mathcal{T}^{(\beta)}$
3. $\forall i \in [s], \ G_i$ is a subgraph of $G$
4. $\forall i, j \in [s], \ i \neq j \Rightarrow X_i \cap X_j = \emptyset$
5. $|V(G) \setminus \bigcup_{i \in [s]} X_i| \leq \alpha$
6. $\forall i \in [s], \ tw(G[X_i]) \leq \gamma$.

We call the set $V(G) \setminus \bigcup_{i \in [s]} X_i$ center of the above $(\alpha, \beta, \gamma)$-protrusion decomposition. Protrusion decompositions have been introduced in [4] in the context of kernelization algorithms (see also [25, 23]). The above definition is a modification of the original one in [4], adapted for the needs of our proofs. The only essential modification is the parameter $\gamma$, used in the last requirement. Intuitively, $\gamma$ bounds the “internal” treewidth of each protrusion $B_i$.

### 2.2 Equivalence on boundaried structures.

**(Counting) Monadic Second Order Logic.** We say that a MSOL-formula $\phi$ is a formula on structures if it has a free variable corresponding to a set of vertices. A structure $G = (G, A)$ is a model for such a formula $\phi$, we write this $G \models \phi$, if it becomes true on $G$ when instantiating the free variable of $\phi$ by the set $A$. Given a MSOL-formula $\phi$ we denote by $|\phi|$ the length of the formula.
Data-Compression for Parametrized Counting Problems on Sparse Graphs

**Equivalences between b-structures and b-graphs.** Let $\phi$ be a MSOL-formula and $t \in \mathbb{N}$. Given two b-structures $G_1, G_2 \in \mathcal{B}^{(t)}$, we say that $G_1 \equiv_{\phi, t} G_2$ if

* $G_1 \sim G_2$
* $\forall F \in \mathcal{B}^{(t)} F \sim G_1 \Rightarrow (F \oplus G_1 \models \phi) \iff (F \oplus G_2 \models \phi)$

Notice that $\equiv_{\phi, t}$ is an equivalence relation on $\mathcal{B}^{(t)}$. The following result is widely known as Courcelle’s theorem and was proven [10]. The same result was essentially proven in [7] and [2]. The version on structures that we present below appeared in [4, Lemma 3.2].

**Proposition 3.** There exists a computable function $\xi : \mathbb{N}^2 \rightarrow \mathbb{N}$ such that for every CMSO-formula $\phi$ and every $t \in \mathbb{N}$, the equivalence relation $\equiv_{\phi, t}$ has at most $\xi(|\phi|, t)$ equivalence classes.

Given a MSOL-formula $\phi$ and under the light of Proposition 3, we consider a (finite) set $\mathcal{R}_{\phi, t}$ containing one minimum-size member from each of the equivalence classes of $\equiv_{\phi, t}$.

Keep in mind that $\mathcal{R}_{\phi, t} \subseteq \mathcal{B}^{(t)}$. Notice that for every $G \in \mathcal{B}^{(t)}$, there is a b-structure in $\mathcal{R}_{\phi, t}$, we denote it by $\text{rep}_{\phi, t}(G)$, such that $\text{rep}_{\phi, t}(G) \equiv_{\phi, t} G$.

### 3 Approximating protrusion decompositions

The main result of this section is a constant-factor approximation algorithm computing a $t$-treewidth modulator (Lemma 5). Based on this we also derive a constant-factor approximation algorithm for a protrusion decomposition (Theorem 6). For our proofs we need the following lemma that is a consequence of the results in [34].

**Lemma 4.** For every $h$-vertex graph $H$ and every $t \in \mathbb{N}$, there exists a constant $c$ and an algorithm that takes as input an $H$-topological-minor-free graph $G$ and a $t$-treewidth modulator $X \subseteq V(G)$ and outputs a $(c|X|, c, t)$-protrusion decomposition along with tree decompositions of its $b$-graphs of width at most $c$, in $O_{h+1}(n)$ steps.

As a consequence of Lemma 4, as long as the input graph $G$ has many vertices (linear in $k$), there is a vertex set $Y$ whose (internal) treewidth is at most $t$ and contains sufficiently many vertices. The key step of the approximation algorithm, to be shown in the next lemma, is to replace $N[Y]$ with a smaller graph of the same ‘type’. Two conditions are to be met during the replacement: first, the minimum-size of a $t$-treewidth modulator remains the same. Secondly, a $t$-treewidth modulator of the new graph can be ‘lifted’ to a $t$-treewidth modulator of the graph before the replacement without increasing the size.

**Lemma 5.** For every $h$-vertex graph $H$ and every $t$, there is a constant $c$, depending on $h$ and $t$, and an algorithm that, given a graph $G \in \mathcal{F}_H$ and $k \in \mathbb{N}$, either outputs an $t$-treewidth-modulator of $G$ of size at most $c \cdot k$ or reports that no $t$-treewidth modulator of $G$ exists with size at most $k$. This algorithm runs in $O_{h+1}(n^2)$ steps.

Notice that the above lemma, with worst running time, is also a consequence of the recent results in [32]. We insist to the above statement of Lemma 5, as we are interested for a quadratic time approximation algorithm for protrusion decompositions. Indeed, based on Lemma 5 we can prove the following that is the main result of this section.

**Theorem 6.** Let $H$ be an $h$-vertex graph and $\phi$ be a MSOL-formula that is treewidth modulable. Then there is a constant $c$, depending on $h$ and $|\phi|$, and an algorithm that, given an input $(G, k)$ of $\Pi_{\phi, F_H}$, either reports no $A \subseteq V(G)$ with $(G, A) \models \phi$ has size at most $k$ or outputs a $(ck, c, c)$-protrusion decomposition of $G$ along with tree decompositions of its $b$-graphs, each of width at most $c$. This algorithm runs in $O_{|\phi|+h}(n^2)$ steps.
4 The compactor

By Theorem 6, we may assume that a \((tk, t, t)\)-protrusion decomposition \(G_1, \ldots, G_s\) of \(G\), with \(G_i = (G_i, B_i, -)\), is given for some \(t\). For counting the sets \(A \subseteq V(G)\) of size at most \(k\) with \((G, A) \models \phi\), we view such a set \(A\) as a union of \(A_0 \cup A_1 \cup \cdots \cup A_s\), where \(A_0\) is the subset of \(A\) residing in the center of the decomposition, and \(A_i = A \cap V(G_i)\) for each \(i \in [s]\). Suppose that \(A'_i \subseteq V(G_i)\) for some \(i \in [s]\) satisfies \((G_i, B_i, A'_i) \equiv_{\phi, t} (G_i, B_i, A_i)\) and \(|A_i| = |A'_i|\). Then, \((A \setminus A_i) \cup A'_i\) has the same size as \(|A|\) and we have \((G, A \setminus A_i \cup A'_i) \models \phi\).

In other words, \(A'_i\) and \(A_i\) are indistinguishable when seen from outside of \(G_i\).

The basic idea of the condenser is to replace all the occurrences of such sets \(A'_i\) (include \(A_i\) itself) with \(O(1)\)-bit information; that is, the number of such sets, the size of \(|A'_i|\), and the equivalence class containing \((G_i, B_i, A'_i)\). Formally, for the given CMSO-formula \(\phi\) and \(t \in \mathbb{N}\), we define the function \(\#sol_{\phi, t}\) so that for each \(R \in R_{\phi, t}\), \(G := (G, B, -) \in \mathcal{T}_{\phi, t}\), we set

\[
\#sol_{\phi, t}(R, G, k) = \left| \{ A \in \left( \begin{array}{c} V(G) \\ k \end{array} \right) \mid R \equiv_{\phi, t} (G, B, A) \} \right|.
\]

This function can be fully computed in linear time on a b-graph of bounded treewidth.

Lemma 7. For every CMSO-formula \(\phi\) and every \(t \in \mathbb{N}\), there exists an algorithm that, given a \(G \in \mathcal{T}_{\phi, t}\) and a tree decomposition of \(G\) of width at most \(t\), outputs \(\#sol_{\phi, t}(R, G, k')\) for every \((R, k') \in R_{\phi, t} \times [0, k]\). This computation takes \(O(|\phi| \cdot \ell(nk^2))\) steps.

The proof of Lemma 7 is based on a dynamic programming procedure. This may follow implicitly from the proofs of Courcelle’s theorem (see [11, 9]).

We are now in position to prove Theorem 1.

Proof of Theorem 1. We describe a polynomial size compactor \((P, M)\) for \(\Pi_{\phi, F_H}\). Given an input \((G, k) \in \mathcal{F}_H \times \mathbb{N}\), the condenser \(P\) of the compactor runs as a first step the algorithm of Theorem 6. If this algorithm reports that there is no set \(A\) of size \(k\) with \((G, k) \models \phi\), the the condenser outputs \(\$\), i.e., \(A(G, k) = \$\). Suppose now that the output is a \((tk, t, t)\)-protrusion decomposition \(G_1, \ldots, G_s\) of \(G\), along with the corresponding tree decompositions, for some constant \(t\) that depends only on \(h\) and \(|\phi|\). Let \(K\) be the boundary of this protrusion decomposition and recall that \(|K|, s \leq tk\). We set \(G_0 = G[K]\) and let \(G_i = (G_i, B_i, -)\) for each \(i \in [s]\). We also define \(B = \{B_i, i \in [s]\}\) where \(B_i\) is the boundary of \(G_i\), \(i \in [s]\). The next step of the condenser is to apply the algorithm of Lemma 7 and compute \(\#sol_{\phi, t}(R, G_i, k')\) for every \((R, k') \in R_{\phi, t} \times [0, k] \times [s]\), in \(O(|\phi| + \ell(nk^2))\) steps. The output of the condenser \(P\) is

\[
P(G, k) = (G_0, B, \{ \#sol_{\phi, t}(R, G_i, k') \mid (R, k', i) \in R_{\phi, t} \times [0, k] \times [s] \}).
\]

Clearly, \(P(G, k)\) can be encoded in \(O(|\phi| + \ell(nk^2))\) memory positions.

We next describe the extractor \(M\) of the compactor. For simplicity, we write \(z := P(G, k)\) and we define \(M(\$) = 0\). We assume that there is a fixed labeling \(\lambda\) of \(G_0\). The extractor \(M\) first computes the set \(A\) containing all subsets of \(K\) of at most \(k\) vertices. Notice that \(|A| = 2^{|\phi| + \ell(k)}\).

Next, for each \(A_0 \in A\), the algorithm builds the set \(M_{A_0}\) containing all mappings \(m : [s] \to R_{\phi, t}\) with the property that, for every \(i \in [s]\), \((G_0, B_0, A_0) \sim m(i)\). As the boundary of \(m(i)\) induces an identical labeled graph as \(B_i\), we denote \(m(i)\) as \((G^m_i, B_i, A^m_i)\). Notice that \(|M_{A_0}| = 2^{|\phi| + \ell(k)}\), for every \(A_0 \in A\).

Let \(A_0 \in A\) and \(m \in M_{A_0}\). For each such pair, the extractor runs a routine that constructs an annotated graph \((D^m, A^m)\) as follows: first it initializes \(D^m_0 = (D_0, A^m_0)\) with \(D_0 = G_0\) and \(A^m_0 = A_0\). After constructing \(D^m_i = (D_i, \bigcup_{j \in [i]} A^m_j)\), the routine
sets $D_{m+1}^i = ((D_i, B_{i+1}, -) \cup (G_m^{i+1}, B_i, -)) \cup (\cup_{j \in [i+1]} A_j)$ iteratively from $i = 0$ up to $s - 1$. We set $(D_m, A_m) = D_0^m$. Notice that the routine runs in $O(|A| + h(k)$ steps and that $|D_m| = O(|A| + h(k)$.

The extractor $M$ is defined as

$$M(z) = \sum_{A_0 \in A} \prod_{m \in M \setminus A_0} [(D_m, A_m) \models \phi] \cdot \left( \sum_{\zeta \in \mathcal{K} - |A_0|} \prod_{i \in [s]} \#\text{sol}_{\phi, \zeta}(m(i), G_i, \zeta(i) + |B_i \cap A_0|) \right)$$

where $[\cdot]$ is a function indicating whether a sentence is true (=1) or false (=0), and $\mathcal{K} - |A_0|$ is the set of all vectors $\zeta \in [0, \ell]^s$ such that $\sum_{i \in [s]} \zeta(i) = \ell - |A_0|$. Having access to $\#\text{sol}_{\phi, \zeta}(R, G, k') \mid (R, k', i) \in \mathcal{R}_{\phi, t} \times [0, \ell] \times [s])$, we can compute $M(z)$ in $2^O(|A| + h(k)$ steps. Therefore, the extractor runs in the claimed running time. It remains to prove that $M(z)$ equals $\{|A \in \binom{V(G)}{k} \mid (G, A) \models \phi\}$.

Before proceeding, we present a key claim (for the proof, see the full version of the paper).

**Claim 8.** Let $H_i = (H_i, B, A_i)$ for $i = 1, 2$ be two compatible $b$-structures from $\mathcal{B}(G)$. Let $H'_2 = (H'_2, B, A'_2)$ be a $b$-structure equivalent with $H_2$. Then for every $B' \subseteq V(H_1)$ of size at most $t$, the two $b$-structures $D$ and $D'$ are equivalent under $\equiv_{\phi, t}$, where

$$D = ((H_1, B, -) \cup (H_2, B, -), B', A_1 \cup A_2) \quad \text{and} \quad D' = ((H_1, B, -) \cup (H'_2, B, -), B', A_1 \cup A'_2)$$

Now, consider an arbitrary sequence $A'_1, \ldots, A'_s$ of vertex sets with $A'_i \subseteq V(G)$, each of which is counted in $\#\text{sol}_{\phi, t}(m(i), G_i, \zeta(i) + |B_i \cap A_0|)$. Claim 8, $[(G, A) \models \phi] = 1$, and $m_i \equiv_{\phi, t} (G_i, B, A')$ ensure that $(G, A_0 \cup \bigcup_{i \in [s]} A'_i) \models \phi$. Observe that

$$|A_0 \cup \bigcup_{i \in [s]} A'_i| = |A_0| + \sum_{i \in [s]} |A'_i \setminus A_i| = |A_0| + \sum_{i \in [s]} |A_i| = |A_0| + \sum_{i \in [s]} \zeta(i) = k.$$  

That is, each combination of $A_0$, $m$, $\zeta$, and a sequence $A'_1, \ldots, A'_s$ contributing 1 to the sum $M(z)$, a vertex set $A$ of size precisely $k$ can be uniquely defined and we have $(G, A) \models \phi$. Clearly, distinct combinations lead to distinct such sets. Therefore, $\{|A \in \binom{V(G)}{k} \mid (G, A) \models \phi\}$ is at least the value of $M(z)$. This completes the proof. $
$  

5 Conclusions

Concerning Theorem 1, we stress that the treewidth-modularity condition can be derived by other meta-algorithmic conditions. Such conditions are minor/contraction bidimensionality and linear separability for graphs excluding a graph/apex graph as a minor [23, 25]. This extends the applicability of our meta-algorithmic result to more problems but in more restricted graph classes. Natural follow-up questions are whether the size of the compactor can be made linear and whether its combinatorial applicability can be extended to more general graph classes.

We envision that the formal definition of a compactor that we give in this paper may encourage the research on data-reduction for counting problems. The apparent open issue is whether other problems (or families of problems) may be amenable to this data-reduction paradigm (in particular, the results in [16, 39, 42, 43] can be interpreted as results on polynomial compactors).

Another interesting question is whether (and to which extent) the fundamental complexity results in [8, 3, 26, 15, 5, 18, 33] on the non-existence of polynomial kernels may have their counterpart for counting problems.
References


Data-Compression for Parametrized Counting Problems on Sparse Graphs


Planar Maximum Matching: Towards a Parallel Algorithm

Samir Datta
Chennai Mathematical Institute & UMI ReLaX, Chennai, India
sdatta@cmi.ac.in

Raghav Kulkarni
Chennai Mathematical Institute, Chennai, India
kulraghav@gmail.com

Ashish Kumar
Chennai Mathematical Institute, Chennai, India
ashishky36@gmail.com

Anish Mukherjee
Chennai Mathematical Institute, Chennai, India
anish@cmi.ac.in

Abstract
Perfect matchings in planar graphs have been extensively studied and understood in the context of parallel complexity [21, 36, 25, 6, 2]. However, corresponding results for maximum matchings have been elusive. We partly bridge this gap by proving:
1. An SPL upper bound for planar bipartite maximum matching search.
2. Planar maximum matching search reduces to planar maximum matching decision.
3. Planar maximum matching count reduces to planar bipartite maximum matching count and planar maximum matching decision.
The first bound improves on the known [18] bound of $LC=L$ and is adaptable to any special bipartite graph class with non-zero circulation such as bounded genus graphs, $K_{3,3}$-free graphs and $K_5$-free graphs. Our bounds and reductions non-trivially combine techniques like the Gallai-Edmonds decomposition [23], deterministic isolation [6, 7, 3], and the recent breakthroughs in the parallel search for planar perfect matchings [2, 32].

2012 ACM Subject Classification Theory of computation → Parallel algorithms

Keywords and phrases maximum matching, planar graphs, parallel complexity, reductions

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.21

1 Introduction

Matchings are one of the most fundamental and well-studied objects in graph theory and in theoretical computer science (see e.g. [23, 20]) and have played a central role in Algorithms and Complexity Theory. Edmond’s blossom algorithm [8] for Maximum-Matching is one of the first examples of a non-trivial polynomial time algorithm. It has had a considerable share in initiating the study of efficient computation, including the class P itself; Valiant’s $#P$-hardness [35] for counting perfect matchings in bipartite graphs provides surprising
Planar Maximum Matching

insights into counting complexity classes. The rich combinatorial structure of matching problems combined with their potential to serve as central problems in the field invites their study from several perspectives.

We consider the following variants of the Maximum-Matching problem. Given $w$, the Decision (or Cardinality) version asks to decide if there is a maximum matching of cardinality at least $w$. The Search and the Counting versions ask respectively for a (witness to a) maximum matching and the number of maximum matchings.

1.1 Parallel Complexity of Matching

The study of whether matching is parallelizable has yielded powerful tools, such as the isolating lemma [29], that have found numerous other applications. The RNC bound remains the best known parallel complexity for Maximum-Matching till date. One of the biggest open problems in this area is to derandomize such construction. Recently, a partial derandomization has put the Perfect-Matching problem in quasi-NC, first for bipartite graphs [10], followed by [34] for general graphs. The best known (non-uniform) upper bound for Perfect-Matching is non-uniform SPL [1].

Matching in Planar and Other Sparse Graphs

A well known example where planarity is a boon is that of counting perfect matchings. The problem in planar graphs is in P [21] and can in fact be placed in NC[36]; thus Perfect-Matching (Decision) in planar graphs is in NC.

In the case of parallel algorithms for planar graphs, the search version seemed harder than the problem of counting. Though the bipartite planar case is known to be in NC[28, 25, 22, 6], the construction version of Perfect-Matching in planar graphs in NC was an outstanding open question and has been solved very recently by Anari and Vazirani [2] and Sankowski [32].

The space complexity of matching problems in planar graphs was first studied in [6] where it is shown that min-weight Perfect-Matching in bipartite planar graphs is in SPL via non-zero circulations. The isolation lemma has also been derandomized for $K_{3,3}$-free and $K_5$-free bipartite graphs, giving the SPL upperbound [3].

However, known results on Maximum-Matching are limited. The only relevant result known to us is computing a maximum matching for bipartite planar graphs in $L^{C=L} \subseteq NC$ by Hoang [18]. A different NC algorithm is given for the same problem in [32]. The related approximation problem has been investigated more. An NC approximation scheme [19] and a Logspace approximation scheme [5] for Maximum-Matching are known for general graphs and classes of sparse graphs (including bounded degree graphs and planar graphs) respectively.

1.2 Maximum Matching and Our Contribution

Since Perfect-Matching is a specialisation of Maximum-Matching, upper bounds applicable for the latter directly translate to the former. Edmond’s blossom-shrinking algorithm and the Micali-Vazirani [27] algorithm fall in this category. Occasionally, it is possible to lift the bounds in the other direction also such as the following:

► Observation 1. Perfect-Matching and Maximum-Matching are equivalent in general graphs under logspace Turing reduction.

Though if we start with a planar graph such reductions does not necessarily keep the graph planar and the goal of this paper is to explore such possibilities for special graph classes.
Recent years have seen considerable progress in upper bounds for Perfect-Matching in planar and other restricted graph classes culminating in [2, 32] which yield efficient parallel algorithms for planar Perfect-Matching Search. In this paper we try to close the gap between perfect and maximum matchings for planar and related graph classes in the context of parallel complexity. Unless otherwise stated, our results hold in planar, bounded genus, $K_{3,3}$-free and $K_5$-free graphs. Our main result is the following.

\textbf{Theorem 2.} Maximum-Matching Search in graphs with non-zero circulation is in $SPL$.

The class $SPL$, prominently studied in [1], consists of languages whose characteristic function is computed by a determinant. The bound improves on the best known upper bound of $L^{C=L}$ by Hoang [18] and matches the known upper bound for bipartite Perfect-Matching for the same classes of graphs [6, 3]. Hoang uses a rank argument whose complexity doesn’t seem to be in $SPL$ - the seemingly best bound being $L^{C=L}$. Instead, we use the standard isolation technique but in a multi-graph (i.e. with self-loops) but make sure that the loop-paths are never optimal and we can focus on the min-weight cycle covers which deterministic isolation helps us find. Since, a deterministic construction of non-zero circulation is known for $K_{3,3}$-free and $K_5$-free bipartite graphs [3] and bounded genus bipartite graphs [7], the result holds for these classes also.

Next, we reduce the problem of finding a maximum matching to determining the size of a maximum matching in the presence of algorithms to (a) find a perfect matching and to (b) solve the bipartite version of the maximum matching, all in the same class of graphs. We use the classic Gallai-Edmonds decomposition theorem for this reduction. Since NC algorithms are now known for Perfect-Matching in bounded genus [2], $K_{3,3}$-free and $K_5$-free graphs [9] then in these classes of graphs using Theorem 2 we get the following:

\textbf{Theorem 3.} Maximum-Matching Search NC-reduces to Maximum-Matching Decision in planar graphs, in bounded genus graphs, in $K_{3,3}$-free graphs and in $K_5$-free graphs.

This shows that, unlike for perfect matching where decision was known to be in NC and the main bottleneck was the search version, for maximum matching the decision problem is the hard cornerstone. Though we are not able to get an NC upper bound for Maximum-Matching, we show that Maximum-Matching Search for the above mentioned classes of graphs is in Pseudo-deterministic NC. Pseudo-deterministic algorithms are probabilistic algorithms for search problems that produce a unique output for each given input except with small probability. That is, they return the same output for all but few of the possible random choices. We call an algorithm pseudo-deterministic NC if it is in RNC, and is pseudo-deterministic. Bipartite Perfect-Matching is known to be in this class [14].

The class of search problems that can be solved in pseudo-deterministic polynomial time was first studied by Goldwasser and Gat [12]. Since then the field of pseudo-determinism has received significant interest, see e.g. [12, 13, 16] with some very recent progress e.g. [31, 14, 17, 15]. As the size of the maximum matching can be found in RNC [29], from Theorem 3 we get that,

\textbf{Theorem 4.} Maximum-Matching Search is in pseudo-deterministic NC for planar graphs, bounded genus graphs, $K_{3,3}$-free graphs and $K_5$-free graphs.

We also consider the counting version of the Maximum-Matching problem. Though we don’t have an NC algorithm even in planar graphs (in fact, to the best of our knowledge it is not even known to be in P), we show that counting maximum matchings in planar graphs NC-reduces to the question in bipartite planar graphs.

The main questions left unanswered in this study are:

Open Question 1. Is Maximum-Matching Decision in planar graphs in NC?

Open Question 2. Is Maximum-Matching Count in bipartite planar graphs in NC?

Organization. After some preliminaries in Section 2, we describe in Section 3 the SPL algorithm for finding maximum matchings in graphs that have non-zero circulations. For bounded genus, $K_{3,3}$-free and $K_5$-free graphs, we give an NC-reduction from the problem of finding a maximum matching to determining the size of a maximum matching, in Section 4. In Section 5, for the same graph classes we show that counting maximum matching NC-reduces to counting maximum matchings in bipartite graphs and determining the size of a maximum matching. We conclude in Section 6 with some open ends.

2 Preliminaries

Let $G = (V, E)$ be an undirected embedded planar graph with $|V| = n$. We sometime think of the edges as bi-directed i.e. they are directed in both the directions. For $e \in E$, let $w(e)$ denote the weight of the edge $e$. A planar graph is a graph that can be embedded in the plane so that no edges cross each other. A graph $G$ is said to have genus $g$ if $G$ has a minimal embedding (an embedding where every face of $G$ is homeomorphic to a disc) on a genus $g$ surface. An $H$-minor free graph $G$ does not contain the graph $H$ as a minor. See standard texts on Graph theory (e.g. [37]) for further information. Consult [30] for definitions and properties of various other sparse graph classes.

A matching in $G$ is a set $M \subseteq E$, such that no two edges in $M$ have a vertex in common. A matching $M$ is called perfect if $M$ covers all vertices of $G$, $M$ of maximum size is called maximum matching. An alternating path is one whose edges alternate between $M$ and $E \setminus M$. We denote the size (the number of edges in the matching) of $M$ by $|M|$ and the weight (sum of the weight of the edges in the matching) by $w(M)$. Size of the maximum matching in $G$ is denoted by $\nu(G)$. We call two edges (also self-loops and multiple edges) of $G$ disjoint, if the set of vertices which are incident on the edges are disjoint. A matching $M$ of $G$ is said to be near-perfect if exactly one vertex of $G$ is not matched in $M$. For a complete treatment on matching see [23].

Complexity Classes. The complexity classes L and NL are the classes of languages accepted by deterministic and non-deterministic logspace Turing machines, respectively. For a non-deterministic Turing machine $M$, let $acc_M(x)$ and $rej_M(x)$ denote the number of accepting and rejecting computations respectively, on an input $x$. Denote $gap_M(x) = acc_M(x) - rej_M(x)$. $GapL$ is the class of functions $f(x)$ such that for some NL machine $M$, $f(x) = gap_M(x)$. A language $L$ is in SPL if so that for all inputs $x$, $gap_M(x) \in \{0, 1\}$ and $x \in L$ if and only if $gap_M(x) = 1$. For a complexity class $C$, we say that a language $L$ $C$-reduces to a language $L'$ if there is a many-one reduction from $L$ to $L'$ computable in the class $C$. NC (RNC) is the class of problems which can be solved using deterministic (randomized) polynomial size circuits of polylogarithmic depth. Define pseudo-deterministic algorithms as follows:
We construct a graph also. This contributes (x, s(x)) ∈ R.

A pseudo-deterministic NC algorithm is an RNC algorithm which is also pseudo-deterministic.

Non-zero Circulation Weights. For any simple cycle $C$ of $G$, we define the circulation of $C$, denoted by $\text{circ}(C)$, as the alternating sum of the weights of the edges of the cycle. Formally, if the cycle is given by $(e_0, e_1, \ldots, e_l)$ then, $\text{circ}(C) = \sum_{i=0}^{l} (-1)^i w(e_i)$. In this paper the classes of graphs, where deterministic weighting schemes are known such that each cycle in the given graph gets non-zero circulation weight, are together referred to as graphs with non-zero circulation.

It is shown in [6] that non-zero circulation weights imply isolating weights for matchings. Also, a simple L-computable weighting function is constructed for grid graphs such that the circulation of every simple cycle is non-zero. In [7] it is shown that using [4] this weighting function can be extended to all bipartite graphs embeddable on a fixed surface. This was further extended to $K_{3,3}$-free and $K_5$-free bipartite graphs in [3].

3 Maximum-Matching Search in graphs with non-zero circulations

In this section we show that given an undirected unweighted graph $G = (V,E)$ admitting non-zero circulations, finding a maximum matching is in SPL. The basic idea is to construct an auxiliary graph $G'$ having the property that finding a maximum matching in $G$ reduces to finding a min-weight generalized perfect matching (defined later) in $G'$. Assign non-zero circulation weights to the edges in $G'$ which are also isolating weights for matchings. Then we extract a min-weight generalized perfect matching from $G'$ which in turn extracts a maximum matching from $G$.

A deterministic construction of non-zero circulation is known in planar bipartite graphs [6], bounded genus bipartite graphs [7] and also in $K_{3,3}$-free and $K_5$-free bipartite graphs [3]. We construct a graph $G' = (V',E')$ from $G$ by adding vertex $t_v$ with a self loop for each vertex $v \in V$ and join $v$ and $t_v$ using an undirected edge, as shown in Figure 1. Thus, $|V'| = 2n \equiv 0 \pmod{2}$. Notice that the genus and the $H$-minor freeness property of $G'$ remains the same as $G$. Define a weight function $w' : E' \mapsto \{0,1\}$ for $G'$ as follows. The original edges of $G$ have weight 1, the self-loops are of weight zero and rest of the new edges have weight 1 (suffices to pick any weight $> 1/2$). We define a generalized matching as a set of disjoint edges (possibly) inclusive of self-loops. Various notions for matching naturally extends to generalized matchings. Call a generalized matching as perfect wherein every vertex is matched and as min-weight perfect if it is perfect and of minimum weight.

Proposition 7. Any matching $M$ in $G$ can be extended to a generalized perfect matching $P$ in $G'$. Moreover, $w'(P) = n - \nu(G')$.

Proof. For each $v \in V$ unmatched in $G$ use the $(v, t_v)$ edge of $G'$ in $P$, thereby matching $t_v$ also. This contributes $(n - 2|M|)$ to $w'(P)$. For the rest of the $2|M|$ vertices $v \in V$ matched...
in $G$, match the corresponding new vertices using the self loop at $t_i$. Since the self loops are of weight zero, the matched edges contribute $|M|$ to $w'(P)$. These form a generalized perfect matching $P$ in $G'$ with $w'(P) = (n - 2|M|) + |M| = n - \nu(G')$.

\[\blacktriangleup\]

**Observation 8.** An extension of a maximum matching in $G$ to $G'$ corresponds to a min-weight generalized perfect matching in $G'$ and a restriction of a min-weight generalized perfect matching of $G'$ to $G$ corresponds to a maximum matching in $G$.

Thus the problem of finding a maximum matching in $G$ is equivalent to that of finding a min-weight generalized perfect matching in $G'$. Now we address the problem of finding isolating weights for extracting a min-weight generalized perfect matching. We define a weight function $w$ (by combining several other weight functions) for which we show the following:

\[\blacktriangleup\]

**Lemma 9.** With respect to the weight function $w : E' \mapsto [N]$, the min-weight generalized perfect matching in $G'$ is unique.

To prove this we need some definitions first. Define a loop-path as a closed trail $(e_0, e_1, e_2, \ldots, e_k)$ (for $k$ odd, $k > 1$) where $e_0$ is a self loop, the subtrail $(e_1, e_2, \ldots, e_{k-1})$ is a path of non-zero length and $e_k$ is also a self loop. Define a 2-cycle as a length 2 directed cycle corresponding to an undirected edge as the underlying graph. Define a 2-self loop as a closed walk $(e, e)$ where $e$ is a self loop. Define the alternating weight of a loop-path $P' = (e_0, e_1, e_2, \ldots, e_k)$ (for $k \geq 2$) to be the alternating sum of the weight of the edges in $P'$. i.e. $AW(P') = \sum_{i=0}^{k} (-1)^i w(e_i) = (w(e_0) - w(e_k)) + (-w(e_1) + w(e_2)) - \ldots + w(e_{k-1})$.

Let the graph $G'$ has at most $c'n$ many edges for some constant $c'$. Define a weight function $w''$ on the edges of $G'$ which assigns non-zero weights to the self-loops as follows,

$$w''(e) = \begin{cases} ic', & \text{if } e = (t_i, t_i) \text{ for } 1 \leq i \leq |V| \\ 0, & \text{otherwise} \end{cases}$$

The non-zero circulation weights of [3], which works for planar, $K_{3,3}$-free and $K_5$-free bipartite graphs, compute the weights for the graph directly. For bounded genus graphs the weighting scheme of [7] work on a grid embedding where they use the weighting scheme of [6] to assign the weights. Following [7], given a graph $H$ whose genus is bounded by some constant, the idea is to create a new graph $H'$ with maximum degree 3 by expanding large degree vertices of $H$ into binary trees preserving the bipartition. Now embed $H'$ onto a constant genus grid $H''$ such that each edge of $H'$ gets expanded into an odd length path in the grid. These are L-reductions preserving the bipartiteness and perfect matchings between $H$ and $H''$ but not maximum matchings. Hence we need to finally pull back the weights assigned in $H''$ to the original graph $H$ ensuring that the non-zero circulation property is preserved.

\[\blacktriangleup\]

**Lemma 10.** The pull-back weights from $H''$ give non-zero circulation to the cycles in $H$ and are polynomially bounded.

Denote this non-zero circulation weight for an edge $e$ by $w'''(e)$ which are bounded by, say $n^c$ for some constant $c$. We combine the weights $w''$ and $w'''$ into a single weight $w^*$. Using bit shift, we define the new weight $w^*(e)$ on the edges of $G'$ by $w^*(e) = w''(e) \cdot 2^{|[c+1] \log_2(n)|} + w'''(e)$ for $e \in E(G')$. The weights $w^*(e)$ are bounded by $w''(e) \cdot n^{c+1} \leq c'n \cdot n^{c+1} \leq c'n^{c+2}$. Notice that for the non-self loop edges $w^*(e)$ is bounded by $w'''(e) \leq n^c$.

\[\blacktriangleup\]

**Lemma 11.** With respect to the weighting scheme $w^*$, the alternating sum of each simple alternating cycle of $G'$ and each loop-path is non-zero.
Proof. Using the weights $w''(e)$ from Lemma 10, each simple alternating cycle of $G$ has non-zero circulation, and since each simple cycle of $G'$ is necessarily a simple cycle in $G$, thus every simple alternating cycle of $G'$ has non-zero circulation. Now consider the loop-path given by $P = (e_0, e_1, e_2, \ldots, e_k)$ (for $k \geq 2$). Then, $|AW(P)| = |\sum_{i=0}^{k}(-1)^i w^*(e_k)| \geq |w^*(e_0) - w^*(e_k)| - |(w^*(e_1) - w^*(e_2) + \ldots + (-1)^k w^*(e_k))|$. And,

$$|(w^*(e_1) - w^*(e_2) + \ldots + (-1)^k w^*(e_k))| < |w^*(e_1)| + |w^*(e_2)| + \ldots + |w^*(e_k)| < (k - 1) \cdot n^{c^k} \quad \text{(as $w^*(e_i) \leq n^{c_k}$ here)}$$

$$< (c' n - 1) \cdot n^{c^k} \quad \text{($k < |E(G')| \leq c' n$)}$$

$$\leq c'n^{c^k + 1}$$

Then $|AW(P)| > |w^*(e_0) - w^*(e_k)| - c'n^{c^k + 1} \geq 0$ and thus every loop-path also has non-zero alternating weight. ▴

Now we combine the weights $w'$ and $w^*$ into a single weight $w$. Using bit shift again, we define the new weight $w(e)$ on the edges of $G'$ as $w(e) = w'(e) \cdot 2^{(c+2)\log_2(c' n)} + w^*(e)$ for $e \in E(G)$. The weights $w(e)$ are bounded by $w'(e) \cdot c'n^{c^k + 2} \leq c'n^{c^k + 2}$ as $w'(e) \in \{0, 1\}$.

Lemma 12. A min-weight generalized perfect matching of $G'$ corresponding to the weight function $w'$ is also a min-weight generalized perfect matching corresponding to the weight function $w$. Moreover, the alternating sum of the weights with respect to $w$ of simple alternating cycles and loop-paths are non-zero.

We are now ready to prove Lemma 9.

Proof of Lemma 9. The components of the superposition of any two generalized perfect matchings are either simple alternating cycles, loop-paths, 2-cycles or 2-self-loops. Suppose that there is more than one min-weight generalized perfect matching of $G'$, call them $P_1$ and $P_2$, such that $P_1 \neq P_2$. Since $P_1 \neq P_2$, there exists at least one component of $P_1 \cup P_2$ which is a simple alternating cycle or an loop-path. And since $w''(e)$ assigns a non-zero alternating sum weight on all simple alternating cycle and loop-paths, this implies that the sum of weights of edges from one of $P_1$ and $P_2$ is lesser than the other. Swapping the edges between $P_1$ and $P_2$ in this component will give rise to a new generalized perfect matching having weight lower than both of $P_1$ and $P_2$, which is a contradiction. ▴

We use the determinant polynomial to compute the size of the maximum matching.

Lemma 13. The union of two generalized perfect matchings of $G'$, whose corresponding maximum matchings on $G$ match a distinct set of vertices, do not appear in the determinant polynomial.

Proof. Since the union of two generalized perfect matchings of $G'$, whose corresponding maximum matchings on $G$ match a distinct set of vertices of $G$ will have an loop-path. Such terms are not represented by any permutation $\sigma$, and do not appear in the summation. ▴

Notice that since the generalized perfect matchings containing a loop-path are of larger weight (from the weights $w''$) than the minimum, we can safely ignore such matchings.

Observation 14. Let $P$ be the unique min-weight generalized perfect matching in $G'$ of weight $W$. Then the least degree term in the determinant polynomial is $x^{2W}$ corresponding to the unique min-weight cycle cover in $G'$ which is a superposition of $P$ with itself.
Now we compute the least degree term in the determinant polynomial using the layered graph method as used in [6]. Start querying from \( i = -c'n^{c+2} \) to \(+c'n^{c+2}\) to find the first term with non-zero coefficient. Once the weight is known, we can extract \( P \) by deleting each edge \( e \) in parallel from \( G' \) and computing the weight of the min-weight generalized perfect matching in \( G' \setminus \{ e \} \). If the weight is unchanged, it implies the edge is not in \( P \), and otherwise it is. Once we have \( P \) in \( G' \), we can find a maximum matching \( M \) in \( G \) using \( M = P \cap E \). Now if \( w(P) = W \), then \( |M| = n - y, \) where \( y = \left\lfloor \frac{W}{2 \log^{c+2}(c' + n/2)} \right\rfloor \). Thus we arrive at the main result of this section:

\[ \textbf{Theorem 15.} \quad \text{Maximum-Matching Search in graphs with non-zero circulation is in SPL.} \]

\textbf{Proof.} Since the edge weights are polynomially bounded, they can be computed in logspace. Moreover computing the coefficient of a term in the determinant polynomial is in \textbf{GapL} [26]. However since we have used isolating weights, the coefficient of the terms starting from \(-c'n^{c+2}\) to the least degree term is either 0 or 1, and hence this computation is in \textbf{SPL} [1]. As a result, we are able to extract a min-weight generalized perfect matching in the graph \( G'\) in \textbf{LSPL = SPL}. Hence, from the Observation 8 and the previous discussion, we can find a maximum matching of the given graph \( G \) in \textbf{SPL}. \[ \square \]

\section{Reduction from Search to Decision}

We reduce the problem of finding a maximum matching to oracle calls for determining the size of a maximum matching in the presence of a parallel algorithm to find a perfect matching and a parallel algorithm to solve the bipartite version of maximum matching.

The reduction uses the classic Gallai-Edmonds theorem (see Theorem 3.2.1 [23]). The crucial observation is based on partition of the vertices given as follows. A vertex \( v \in V(G) \) belongs to the set of “deficient” vertices \( D(G) \) if there exists some maximum matching of \( G \) that leaves \( v \) unmatched. A vertex \( v \in V(G) \) belongs to \( A(G) \), the set of vertices “adjacent” to \( D(G) \) if \( v \) is a neighbour of some vertex \( u \in D(G) \) and \( v \notin D(G) \). Rest of the vertices in \( V(G) \setminus (D(G) \cup A(G)) \) are in the “critical” set \( C(G) \). A graph \( G \) is said to be factor-critical if for every \( v \in V(G) \), \( v(G) = v(G - v) \).

\[ \textbf{Theorem 16 (Gallai-Edmonds).} \quad \text{Let} \ G \ \text{be a graph and} \ D(G), A(G), \ C(G) \ \text{are defined as above then the components of} \ D(G) \ \text{are factor-critical and every maximum matching in} \ G \] is a perfect matching on \( C(G) \), is near-perfect matching on each component of \( D(G) \), and matches each vertex in \( A(G) \) to a distinct component in \( D(G) \).

\[ \textbf{Observation 17.} \quad \text{For a vertex} \ v \in V(G), \ v \in D(G) \ \text{if and only if} \ v(G) = v(G - v). \]

Next we can find if \( v \in A(G) \) if and only if \( v \notin D(G) \) and there is a vertex \( u \in D(G) \) such that \( v \in N(u) \). Finally, \( C(G) = V(G) \setminus (D(G) \cup A(G)) \). From the statement of the Gallai-Edmonds theorem it suffices to find:

1. A perfect matching in each connected component of \( C(G) \).
2. A maximum matching in the bipartite graph formed by contracting each component of \( D(G) \) into a single vertex \( d \) and adding an edge to each vertex \( a \in A(G) \) such that the corresponding component had an edge to \( a \).
3. A perfect matching in each component of \( D(G) \) minus an arbitrary vertex.

\[ \textbf{Lemma 18.} \quad \text{For any class of graphs closed under vertex deletions and edge contractions, there is an NC algorithm for Maximum Matching Search in the class, with oracle queries to Maximum-Matching Size, Maximum-Bipartite-Matching Search and Perfect-Matching Search all for the same class of graphs.} \]
Then from the recent breakthrough works for NC algorithms for perfect matching respectively in bounded genus \([2]\) and in \(K_{3,3}\)-free and \(K_5\)-free graphs \([9]\) along with our maximum matching algorithm for bipartite graphs from Section 3, we obtain the following,

\[\text{Corollary 19. Maximum-Matching Search NC-reduces to Maximum-Matching Decision in planar graphs, in bounded genus graphs, in } K_{3,3}\text{-free graphs and in } K_5\text{-free graphs.}\]

From the above result we also get a pseudo-deterministic NC algorithm for Maximum-Matching Search in the same class of graphs. Recall that pseudo-deterministic algorithms are probabilistic algorithms for search problems that produce a unique output for each given input except with small probability. Since size of the maximum matching can be found in RNC \([29]\) from the above result we get that,

\[\text{Theorem 4 (Restated). Maximum-Matching Search is in pseudo-deterministic NC for planar graphs, bounded genus graphs, } K_{3,3}\text{-free graphs and } K_5\text{-free graphs.}\]

5 Reducing Count to Bipartite Count

We reduce the problem of counting the number of maximum matchings in a (possibly non-bipartite) graph to oracle calls for counting maximum matchings in bipartite graphs in the presence of a parallel algorithm to count the number of perfect matchings. We do this via a two step process: first reduce the problem of counting maximum matchings in the given graph to counting maximum weight matchings in a bipartite graph and then subsequently reducing the problem of counting of maximum weight matching to counting maximum cardinality matchings while the graph remains bipartite.

This reduction again uses the Gallai-Edmonds decomposition theorem. Recall that, in the decomposition, the vertices in \(C(G)\) have a perfect matching and each of the component of \(D(G)\) is factor-critical. So we have that the count of maximum matchings in \(G\) is the product of the count of the perfect matchings in \(C(G)\) and the count of the maximum matchings in \(G \setminus C(G) = A(G) \cup D(G)\).


\textbf{Proof.} Let \(D_1, D_2, \ldots, D_k\) be the components of \(D(G)\). Replace each edge \((a, d)\) between a vertex \(a \in A(G)\) and a vertex \(d \in D_i\), by adding a weight equal to the number of perfect matchings in the component \(D_i \setminus d\). Next we contract each component of \(D(G)\) into a single vertex \(d\). Replace all the parallel edges between \(a\) and \(d\) (created due to the contraction) with a single edge \((a, d)\) of weight equal to the sum of weights on the corresponding parallel edges. Since from the Gallai-Edmonds theorem we know that no maximum matching contains an edge between any two vertices of \(A(G)\), we have ourselves the weighted bipartite graph instance \(G'\). It is easy to see that the number of maximum matchings in \(A(G) \cup D(G)\) equals to the sum \(\sum_{M \in \mathcal{M}(G')} \omega(M)\) where \(\mathcal{M}(G')\) is the set of maximum weighted matchings in \(G'\). This completes the first part of the reduction. In the presence of an oracle access to Perfect-Matching Count the construction is easily seen to be in L.

\[\text{Gadget Construction.}\] We now replace the weighted bipartite graph \(G'\) by an unweighted instance \(G''\) while keeping the counts same. Notice that the the count of the perfect matchings and hence the edge weights takes polynomial (in \(n\)) bits, say \(\ell\) bits, to store. For \(a \in A(G')\) and \(d \in D(G')\), let the weight of the edge \((a, d)\) be \(w(a, d)\) and let \(b_1 b_2 b_3 \ldots b_{\ell}\)
be the binary expansion of $w(a,d)$ i.e. $w(a,d) = b_1 2^{\ell-1} + b_2 2^{\ell-2} + \ldots + b_\ell 2^0$. Equivalently, $w(a,d) = \sum_{i \in S} 2^{|i|-1}$ where the set $S$ is the set of indices corresponding to the non-zero bits in the binary expansion of $w(a,d)$. We replace the edge $(a,d)$ by the gadget $G_{a,d}$ where $a$ and $d$ are connected by $|S|$ many disjoint paths where the $i$-th path, which corresponds to the bit $b_i$ and is present iff $b_i = 1$ (i.e. these paths are indexed by $1$ to $\ell$), is of length $2(\ell - i) + 2$.

Consider the $i$-th path. Call the vertices adjacent to $a$ and $d$ as $a_{i0}$ and $d_{i0}$, respectively. Call the rest of the $2(\ell - i)$ vertices on the path as $x_{ij}, y_{ij}$ alternately for $1 \leq j \leq \ell - i$ where $x_{i1}$ is attached to $a_{i0}$ and $y_{i(\ell-i)}$ is attached to $d_{i0}$. For the $i$-th path, add $2(\ell - i)$ new vertices and call them as $x_{ij}', y_{ij}'$ alternately for $1 \leq j \leq \ell - i$. Add the undirected edges $(x_{ij}, x_{ij}'), (x_{ij}', y_{ij})$, $(y_{ij}, y_{ij}')$ for all $i \in S$ and $1 \leq j \leq \ell - i$. Each such modified path is informally called as box-path. Connect $x_{i1}$ and $y_{i(\ell-i)}$ with a separate path $x_{i1}, a_{i1}, d_{i1}, y_{i(\ell-i)}$ of length $3$ where each consecutive vertex has an edge between them. See Figure 2 where we assume $b_1 = 1$. Notice that the graph remains bipartite after attaching the gadgets.

**Lemma 21.** The gadget $G_{a,d}$ has the following properties:

1. There is a unique perfect matching in $G_{a,d} \setminus \{a,d\}$.
2. If $a$ is matched inside $G_{a,d}$ then $G_{a,d}$ has a perfect matching. If $a$ is matched with a vertex outside $G_{a,d}$, then $G_{a,d} \setminus \{a\}$ has a near-perfect matching.
3. The vertices $a$ and $d$ remain in different bipartitions. A vertex $v \in G_{a,d}$ is in $A(G'')$ if only if it is in the same bipartition as $a$. Rest of the vertices are in $D(G'')$.
4. There are $3w(a,d)$ many maximum (either perfect or near-perfect) matchings in $G_{a,d}$.

Recall that, in a maximum matching in the weighted graph $G''$ all the $A(G'')$ edges are matched. Since for each edge $(a,d)$ in the matching we get a factor $3$ extra in the count than the desired $w(a,d)$ (notice that these weights are multiplicative), we finally divide the count of the maximum matchings in the bipartite unweighted graph $G''$ by $3^{(|A(G'')|)}$ to get the correct count. For any $w \in \mathbb{N}$ we can construct such a gadget and replace every non-unit weight edge of $G'$ by the gadget of the corresponding weight. This completes the second part of the reduction. Since other than counting perfect matchings all the steps of the reduction can be done in $L$, we have that:


A modification of the result of [11] combined with the techniques from [24] gives an NC algorithm for counting perfect matchings in logarithmic genus graphs [25]. Vazirani [36] and Straub et al. [33] show that counting perfect matchings is in NC in $K_{3,3}$-free graphs and $K_5$-free graphs respectively. And hence, along with Theorem 22 we have the following result,

6 Conclusion and Open Ends

The main contribution of this investigation is a better complexity bound on bipartite planar maximum matching which matches the upper bound for bipartite planar perfect matching. We also show an NC reduction from planar maximum matching search to planar maximum matching decision along with an NC reduction from counting planar maximum matchings to counting bipartite planar maximum matchings and planar maximum matching decision (where the NC-bounds hide the complexity of finding and counting planar perfect matching, respectively). To reiterate, the main open questions are to find NC algorithms to determine the size of planar maximum matching and for counting bipartite planar maximum matchings.

References

Planar Maximum Matching


Distributed Approximation Algorithms for the Minimum Dominating Set in $K_h$-Minor-Free Graphs

Andrzej Czygrinow
School of Mathematical and Statistical Sciences, Arizona State University, Tempe, AZ, 85287-1804, USA
aczygr@asu.edu

Michał Hanckowiak
Faculty of Mathematics and Computer Science, Adam Mickiewicz University, Poznań, Poland mhanckow@amu.edu.pl

Wojciech Wawrzyniak
Faculty of Mathematics and Computer Science, Adam Mickiewicz University, Poznań, Poland
wwawrzy@amu.edu.pl

Marcin Witkowski
Faculty of Mathematics and Computer Science, Adam Mickiewicz University, Poznań, Poland
mw@amu.edu.pl

**Abstract**

In this paper we will give two distributed approximation algorithms (in the Local model) for the minimum dominating set problem. First we will give a distributed algorithm which finds a dominating set $D$ of size $O(\gamma(G))$ in a graph $G$ which has no topological copy of $K_h$. The algorithm runs $L_h$ rounds where $L_h$ is a constant which depends on $h$ only. This procedure can be used to obtain a distributed algorithm which given $\epsilon > 0$ finds in a graph $G$ with no $K_h$-minor a dominating set $D$ of size at most $(1 + \epsilon)\gamma(G)$. The second algorithm runs in $O(\log^*|V(G)|)$ rounds.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Distributed algorithms, minor-closed family of graphs, MDS

1 Introduction

The minimum dominating set (MinDS) problem is one of the classical graph-theoretic problems which is of theoretical and practical importance. A subset $D$ of the vertex set in a graph $G$ is called a dominating set in $G$ if every vertex of $G$ is either in $D$ or has a neighbor in $D$. In the minimum dominating set problem the objective is to find a dominating set $D$ of the smallest size. In this paper we will study distributed approximation algorithms in the Local model for the MDS problem in $K_h$-minor-free graphs.

Although the MDS problem is NP-complete even in planar graphs, there are efficient approximation algorithms. Significant progress has been made in recent years in understanding distributed complexity of many classical graph-theoretic problems in some classes of sparse graphs. In the case of the maximum independent set problem, (MaxIS Problem), it is

1 Research supported in part by Simons Foundation Grant # 521777.
known ([2]) that finding deterministically a constant approximation of $\alpha(G)$ even in the case when $G$ is a cycle on $n$ vertices requires $\Omega(\log^* n)$ rounds. At the same time, it is possible to find an independent set in a planar graph $G$ of size at least $(1 - \epsilon)\alpha(G)$ in $O(\log^* n)$ rounds ([2]). In fact, this result immediately extends to graphs $G$ with no $K_h$-minor using the same approach. Even more can be proved for the maximum matching problem (MaxM Problem). On one hand, the above lower bound for the independence number extends to matchings and on the other hand, there is a distributed algorithm which finds in $O(\log^* n)$ rounds a matching $M$ of size at least $(1 - \epsilon)\beta(G)$ even in graphs of bounded arboricity ([1]). This procedure relies on augmenting paths and is very specific to the maximum matching problem. At the same time, the lower bound for the approximation of maximum independent set, does not extend to the minimum dominating set problem, and a constant-approximation which runs in a constant number of rounds is known for planar graphs and graphs with bounded genus. Specifically, Lenzen et. al. gave in [4] a distributed algorithm which in $O(1)$ rounds finds a dominating set of size at most $126\gamma(G)$ in a planar graph $G$ and Amiri et. al. ([6]) gave $O(g)$-approximation for graphs of genus bounded by $g$ which runs in $O(1)$ rounds. The landscape changes when randomization is allowed. It can be shown that there is a randomized algorithm which in $O(1)$ rounds finds with high probability an independent set $I$ of size at least $(1 - \epsilon)\alpha(G)$ in $O(1)$ rounds in a planar graph $G$ ([2]) and similar results can be obtained for the maximum matching. In addition, Lenzen and Wattenhofer [3] showed that there is a $O(a^2)$-approximation of a minimum dominating set can be found in the randomized time $O(\log \Delta)$ in a graph of arboricity $a$.

In this paper, we will propose deterministic distributed approximation algorithms for the MinDS problem in $K_h$-minor-free graph.

Recall that $H$ is called a minor of $G$ if $H$ can be obtained from a subgraph of $G$ by a sequence of edge contractions. A graph is called $K_h$-minor-free if it doesn’t contains the complete graph $K_h$ as a minor. For a graph $H$, $TH$, a topological copy of $H$, is a graph obtained from $H$ by subdividing each edge $e \in H$ $l_e$ times, for some $l_e \in N_0$. Many important classes of graphs, like for example planar graphs, graphs of bounded genus, or bounded tree-width are $K_h$-minor-free for some $h$. As a result our work on $K_h$-minor-free graphs generalizes previous results on planar and bounded-genus graphs.

Our algorithms work in the Local model. This is a synchronous model where a network is modeled as an undirected graph. Each vertex corresponds to a computational unit and an edge to a communication link between two units. Computations proceed in synchronous rounds and in each round a vertex can send and receive messages from its neighbors and can perform local computations. Neither the amount of local computations nor the size of messages is restricted in any way. In addition, we shall assume that vertices of $G$ have unique identifiers from $\{1, \ldots, n\}$, where $n$ is the order of $G$. Consequently, in the case of the MinDS problem, for an underlying network $G$ the objective is to find a set $D \subseteq V(G)$, in the above model, which is a minimum dominating set and has size $O(\gamma(G))$.

We will prove two results on distributed algorithms for the minimum dominating set problem. Our main theorem shows that in the case of graphs $H$ which no $TK_h$ it is possible to find a constant approximation of $\gamma(H)$ in $O(1)$ rounds. Specifically, we have the following theorem.

**Theorem 1.** Let $h \geq 2$. There exists a distributed algorithm which in a graph $H$ of order $n$ which has no $TK_h$ finds in $L_h$ rounds a dominating set $D$ such that $|D| \leq C_h \gamma(H)$, where $L_h$ and $C_h$ are depending on $h$ only.

We didn’t try to optimize constants $C_h$ and $L_h$ and especially in the case of $C_h$ our proof gives a very big constant. In addition, its value depends on the constant from one of the facts from [5] (see Lemma 3), which in turn, depends on $h$. 

"
Theorem 1 generalizes results for planar graphs from [4] and bounded genus graphs from [6] to graphs with no topological copy of $K_h$. In addition, it gives a deterministic $O(1)$-approximation of the MinDS Problem which runs in a constant number of rounds in an important subclass of graphs of bounded arboricity. The proof of Theorem 1 is split into two main steps. In the first step, an algorithm finds certain partitions of $N(v)$ for vertices $v$, and in the second, for every set $W$ in these partitions, $W$ finds a vertex $v$ such that $W \subseteq N(v)$ and $v$ is a “safe” choice to be added to a dominating set. The proof uses a fact from [5], in addition with some new ideas.

Clearly, if $G$ is a graph which is $K_h$-minor-free then it contains no $TK_h$. Consequently, Theorem 1 can also be used, in connection with methods from [2], to obtain a much better approximation ratio in $O(\log^* n)$ rounds when restricted to graphs with no $K_h$-minor.

\textbf{Theorem 2.} Let $h \geq 2$. There exists a distributed algorithm which given $\epsilon > 0$ finds in a $K_h$-minor-free graph $H$ of order $n$ a dominating set $D$ such that $|D| \leq (1 + \epsilon)\gamma(H)$. The algorithm runs $C \log^* n$ rounds where $C$ depends on $h$ and $\epsilon$ only.

Basically, to prove Theorem 2, we first find a constant approximation of $\gamma(H)$ in $H$ using Theorem 1 and then apply a procedure from [2] to find a better approximation. This in turn, generalizes a corresponding result from [2] to graphs which are $K_h$-minor free. Note that once Theorem 1 is established, Theorem 2 can be proved by appealing to a fact from [2] which extends to graphs with no $K_h$-minors in a straightforward way and the rest of the paper is focused on proving Theorem 1. On the other hand, proving Theorem 1 requires new observations, we will discuss our main tool of building certain partitions of sets $N(v)$ arising from the so-called pseudo-covers. In Section 3 we will prove Theorem 1.

\section{Preliminaries}

In this section, we will introduce auxiliary concepts which are used in our algorithm. We will start with some definitions. Let $G = (V, E)$ be a graph. For two sets $A, B \subseteq V$, an $A, B$-path is a path which starts in a vertex from $A$, ends in a vertex from $B$ and has all internal vertices from $V \setminus (A \cup B)$. In the case $A = \{a\}$, we will simplify the notation to an $a, B$-path.

Let $D \subseteq V$ and let $v \in V \setminus D$. A $v, D$-fan is a set of $v, D$-paths $P_1, \ldots, P_s$ such that for $i \neq j$, $V(P_i) \cap V(P_j) = \{v\}$. For $k, l \in \mathbb{Z}^+$ and set $D$, let $D_{k,l}$ be the set of vertices $w$ such that there is $w, D$-fan consisting of $k$ paths each of length at most $l$. We have the following fact from [5].

\textbf{Lemma 3.} For $h, l \in \mathbb{Z}^+$ there is $c$ such that the following holds. Let $G$ be a graph with no $TK_h$ and let $D$ be a dominating set in $G$. Then $|D_{h-1,l}| \leq c|D|$.

To describe the intuition behind our approach for approximating a minimum dominating set $D^*$ consider a planar graph $G$. Let $v \in G$ be an arbitrary vertex. Then $N(v)$ is dominated by some vertices from $D^*$. It is possible that the minimum number of vertices needed to dominate $N(v)$ is “big”, in this case, in view of Lemma 3 (with $l = 1$), adding $v$ or any other vertex of a similar type, will yield a dominating set of size $O(|D^*|)$. Consequently, we have to address the case when the number of such vertices in $D^*$ is small. The main idea of how to build on this intuition is as follows. Supposing $|N(v)| \geq 3$ it is not possible to have three
different vertices which dominate the whole set \( N(v) \), as it gives a copy of \( K_{3,3} \). A similar fact should be true if instead of three vertices dominating \( N(v) \), we have for some constant \( t \) sets \( U_1, \ldots, U_t \) such that the size of each each \( U_i \) is constant and vertices from \( U_i \) dominate \( N(v) \). This however is not exactly true. Indeed, for example, it is possible that there is one vertex \( u \) dominating all but one vertex from \( N(v) \) and many vertices \( v_1, \ldots, v_s \), each dominating the remaining vertex from \( N(v) \), so that \( U_1 = \{ u, v_i \} \). However the contribution, i.e. the number of vertices covered in \( N(v) \), by each \( v_i \) is minimal and, as it turns out, it can be ignored. Building on this, our approach is to find for \( N(v) \), a family of the so-called pseudo-covers, that is sets of a constant size which cover almost all vertices from \( N(v) \) and such that each vertex makes a substantially contribution. Note that \( \{ v \} \) is a choice for such a pseudo-cover. We will argue that the number of such pseudo-covers must be constant and will use these covers to partition \( N(v) \) into a constant number of sets of which each, but the exceptional class, will be big and will be covered by a constant number of vertices. Of course, it is not clear which of these vertices should be included in a dominating set, but suppose initially that there are only two vertices \( v \) and \( u \) which cover a set in this partition. We claim that adding \( u \) is a reasonable choice. As indicated above, we will be able to assume that one of \( u \) and \( v \) is in \( D^* \). If we are lucky then \( u \in D^* \); if however \( v \in D^* \), then since there is a constant number of vertices in pseudo-partitions, adding \( u \) or any other vertex because of \( v \) yields a constant approximation.

Describing these ideas more formally requires a little bit of preparation. Let \( G = (V, E) \) be a graph. We say that \( Z \subseteq V \) is a cover of \( W \subseteq V \) if \( Z \cap W = \emptyset \) and \( W \subseteq \bigcup_{x \in Z} N(x) \).

Let \( W \subseteq V \) and let \( x \in V \setminus W \). We say that \( x \) is \( \alpha \)-strong for \( W \) if \( |N(x) \cap W| \geq \alpha |W| \). Using the same idea as in a proof of the Kővári-Sós-Turán theorem we have the following fact.

**Fact 4.** Let \( \alpha \in (0, 1) \), \( t, s \in \mathbb{Z}^+ \) and let \( M := \frac{(t-1)s^e}{\alpha^d} \). If \( G = (V, E) \) is a graph with no \( K_{t,s} \) and \( W \subseteq V \) is such that \( |W| \geq s/\alpha \), then there are at most \( M \alpha \)-strong vertices in \( V \setminus W \) for \( W \).

**Proof.** Let \( U \) denote the set of \( \alpha \)-strong vertices for \( W \). We will count the number of claws \( K_{1,s} \) in the graph \( G[U, W] \) with centers in \( U \). On one hand, the number of claws is at least \( |U| \left( \frac{\alpha |W|}{s} \right) \), and on the other hand, since \( G[U, W] \) has no \( K_{t,s} \), every \( s \)-element subset of \( W \) can be involved in at most \( t - 1 \) claws. Thus

\[
|U| \left( \frac{\alpha |W|}{s} \right) \leq \left( \frac{|W|}{s} \right) (t - 1)
\]

and so

\[
|U| \left( \frac{\alpha |W|}{s} \right)^s \leq \left( \frac{|W|}{s} \right)^s (t - 1)
\]

which gives \( |U| \leq \left( \frac{(t-1)s^e}{\alpha^d} \right) \).

We are now ready to define the main concept which is used in our algorithm, the notion of an \((\alpha, q, l, K)\)-pseudo-cover.

**Definition 5.** An \((\alpha, q, l, K)\)-pseudo-cover of a set \( W \subseteq V \) is a vector of vertices \((x_1, \ldots, x_m)\) such that for every \( i, x_i \notin W \), and the following conditions are satisfied.

(a) \( |W \setminus \bigcup_{i=1}^m N(x_i)| \leq q; \)

(b) \( x_i \) is \( \alpha \)-strong for \( W \setminus \bigcup_{j<i} N(x_j) \);

(c) \( |N(x_i) \cap (W \setminus \bigcup_{j<i} N(x_j))| \geq l; \)

(d) \( m \leq K \).
When using the concept, \( \alpha \) will be a constant from \((0, 1)\), and \( q, l, K \) will be constants which depend on \( h \) when we consider graphs with no \( TK_K \). To be more precise,

\[
K := 2h - 2, \quad \alpha := \frac{1}{K}, l := \frac{h}{\alpha} + 1, q := K \cdot l.
\]

In addition, we will have

\[
s := h, t := \left(\frac{h}{2}\right) + h.
\]

Also note, that in the degenerate case when \( |W| \leq q \), we will allow the empty vector.

It is not difficult to see that any cover of a set \( W \) with at most \( K \) vertices contains an \((\alpha, q, l, K)\)-pseudo-cover with \( \alpha = 1/K \) and \( l = q/K \).

\textbf{Fact 6.} For every \( q \) and every cover \( Z \) of \( W \) such that \( |Z| \leq K \) there is an ordering of vertices of \( Z, (x_1, \ldots, x_K) \), such that for some \( m \leq K, (x_1, \ldots, x_m) \) is an \((\alpha, q, l, K)\)-pseudo-cover of \( W \) with \( \alpha = 1/K \) and \( l = q/K \).

\textbf{Proof.} Let \( l := q/K \). If \( |W| \leq q \), then the pseudo-cover is empty. Otherwise, let \( x_1 \in Z \) be such that \( |N(x_1) \cap W| \) is maximum. Then \( |N(x_1) \cap W| \geq |W|/K \geq l \). For the general step. Suppose \( |W \setminus \bigcup_{j<i} N(x_j)| > q \). Then there exists \( y \in Z \setminus \{x_1, \ldots, x_{i-1}\} \) such that \( |N(y) \cap (W \setminus \bigcup_{j<i} N(x_j))| \geq |W \setminus \bigcup_{j<i} N(x_j)|/K \). Set \( x_i := y \). We have \( |N(y) \cap (W \setminus \bigcup_{j<i} N(x_j))| > q/K \). ▲

One of the key observations used in the proof is that the number of \((\alpha, q, l, K)\)-pseudo-cover of a set \( W \) does not depend on \( |W| \).

\textbf{Lemma 7.} Let \( \alpha \in (0, 1) \) be such that \( s/\alpha \) and \( q := l \cdot K \). Then for every graph \( G \) with no \( K_{s,t} \) and every \( W \subseteq V(G) \) such that \( |W| \geq l \), the number of \((\alpha, q, l, K)\)-pseudo-covers of \( W \) is at most \( 2 \left( \frac{(u-1)c^e}{\alpha^e} \right)^K \).

\textbf{Proof.} Suppose the number of \((\alpha, q, l, K)\)-pseudo-covers is bigger than \( C := 2 \left( \frac{(u-1)c^e}{\alpha^e} \right)^K \).

Since the first positions are \( \alpha \)-strong for \( W \) and \( |W| \geq s/\alpha \), by Fact 4 there can be at most \( M := \frac{(u-1)c^e}{\alpha^e} \) of \((\alpha, q, l, K)\)-pseudo-covers with distinct first positions. Let \( x_1 \) be a vertex which appears most often in the first position of these covers. Then at least \( C/M > 1 \) of the covers have \( x_1 \) in the first position and out of these there can be at most one \((\alpha, q, l, K)\)-pseudo-cover which contains only \( x_1 \). If \( |W \setminus N(x_1)| \leq l \) then no vertex can cover more than \( l \) vertices of \( W \setminus N(x_1) \). Thus we may assume otherwise. Now we can iterate the above argument restricting attention to those \((\alpha, q, l, K)\)-pseudo-covers which have \( x_1 \) in the first position. We have \( |W \setminus N(x_1)| > l > s/\alpha \), and so by Fact 4 at least \( (C/M - 1)/M = (C - M)/M^2 > 1 \) vectors have the second position equal to some \( x_2 \). Iterating the above gives that there are at least

\[
(C - M - M^2 - \cdots - M^{i-1})/M^i
\]

\((\alpha, q, l, K)\)-pseudo-covers starting with \( x_1, x_2, \ldots, x_i \) for some \( x_1, \ldots, x_i \). Since a pseudo-cover has at most \( K \) vertices, \( C < 2MK \) for the above quantity to be at most one when \( i = K \); a contradiction. ▲

Let \( v \) be such that there exist \( K \) vertices \( x_1, \ldots, x_K \in V \setminus \{v\} \) with the property \( N(v) \subseteq \bigcup_{k \leq K} N(x_k) \). The number of such covers of \( N(v) \) can be “large” but in view of Fact 6 and Lemma 7 the number of \((\alpha, q, l, K)\)-pseudo-covers such that \( l > s/\alpha \) obtained from covers of
N(v) is a constant independent of |N(v)|. In the rest of the section we will use the fact that the number of \((\alpha, q, l, K)\)-pseudo-covers is constant to refine partitions determined by the covers into a constant number of sets. Fix \(0 < \alpha < 1\) and \(s, K \in \mathbb{Z}^+\) and \(l\) so that \(l > s/\alpha\). Let \(v\) be such that \(|N(v)| \geq l\).

Let \(T(v)\) denote the set of \((\alpha, q, l, K)\)-pseudo-covers of \(N(v)\). By Lemma 7, we have |

\[
|T(v)| \leq C
\]

where \(C := 2 \left( \frac{(1-\alpha)e^\alpha}{\alpha} \right)^K\). For \(S := (x_1, \ldots, x_m) \in T(v)\) consider the following partition \(P_S = \{W_0, W_1, \ldots, W_m\}\) of \(N(v)\). Let \(W_1 := N(x_1) \cap N(v), W_i := (N(x_i) \cap N(v)) \setminus \bigcup_{j \leq i} W_j\) for \(i > 1\), and let \(W_0 := N(v) \setminus \bigcup_{j \leq m} N(x_j)\). Since \(S\) is an \((\alpha, q, l, K)\)-pseudo-cover of \(N(v)\), we have \(|W_0| \leq q\).

Let \(Q(v)\) be the minimal partition which refines partitions \(P_S\) over all \((\alpha, q, l, K)\)-pseudo-covers \(S\) from \(T(v)\). For example, if there are only two partitions, \(P_S = \{W_0, W_1, \ldots, W_m\}\) and \(P_T = \{U_0, U_1, \ldots, U_m\}\), then \(Q(v)\) contains all non-empty intersections \(W_i \cap U_j\).

**Fact 8.** \(|Q(v)| \leq 2^{(K+1)C}\)

**Proof.** For \(S = (x_1, \ldots, x_m)\), we have \(|P_S| \leq m + 1 \leq K + 1\) and so there are at most \((K+1)C\) different subsets of \(N(v)\) over all \(S \in T(v)\). Taking the refinement of these partitions results in at most \(2^{(K+1)C}\) sets.

We will now modify \(Q(v)\) as follows. Let \(V_0\) be the union of these partition classes in \(Q(v)\) which are subsets of \(W_0 = N(v) \setminus \bigcup_{j=1}^m N(x_j)\) for at least one \(S = (x_1, \ldots, x_m)\). Let \(\{V_1, \ldots, V_s\}\) denote the remaining partition classes. Then \(\{V_0, V_1, \ldots, V_s\}\) is a partition of \(N(v)\) (See Figure 1 for an illustration). In addition, we have the following fact.

**Fact 9.** The following conditions are satisfied.

1. \(|V_0| \leq Cq\).
2. For \(i \geq 1\) and for every \((x_1, \ldots, x_m) \in T(v)\), \(V_i \subset N(x_j)\) for some \(j \in [m]\).

**Proof.** The number of vertices which belong to at least one set \(W_0\) is at most \(Cq\). For part (2), fix \(V_i\) for \(i \geq 1\) and let \((x_1, \ldots, x_m) \in T(v)\). Then vertices from \(V_i\) are covered by \(\bigcup_{j=1}^m N(x_j)\) because \(V_i\) doesn’t intersect \(N(v) \setminus \bigcup_{j=1}^m N(x_j)\). Let \(j_1\) be the smallest index such that \(V_i \cap N(x_{j_1}) \neq \emptyset\) and suppose that for some \(j_2 > j_1\), we have \(V_i \cap (N(x_{j_1}) \setminus N(x_{j_2})) \neq \emptyset\). Then \(V_i\) intersects \(W_{j_2}\), but since it is not contained in \(W_{j_2}\), it intersects another set in the \((\alpha, q, l, K)\)-pseudo-cover determined by \((x_1, \ldots, x_m)\), and so it cannot belong to \(\{V_0, V_1, \ldots, V_s\}\).
We will end this section with some more notation which will be used later. Recall that $T(v)$ denotes the set of $(\alpha,q,l,K)$-pseudo-covers of $N(v)$. For set of vertices $U$ we will define $T(U) := \bigcup_{v \in U} T(v)$. For a set $S$ of $(\alpha,q,l,K)$-pseudo-covers we let $V_S$ be the set of vertices which belong to at least one pseudo-cover from $S$. We will slightly abuse above notation and define $T(S) := T(V_S)$.

Using the above convention, we will use $T^{(i)}(U) := T(T^{(i-1)}(U))$ with $T^{(1)}(U) := T(U)$ and $T^{(\leq k)}(U) := \bigcup_{1 \leq i \leq k} T^{(i)}(U)$.

## 3 Algorithm

In this section we will give the main algorithm. The algorithm consists of two phases. In the first phase we simply add to a dominating set $D$ vertices $v$ which have only one vector in $T(v)$, namely $(v)$. In the second phase, we analyze sets in $Q(v)$ and argue that if a set $V_i$ is big enough then we will be able to find a “good” choice among a constant number of vertices from vectors in $T(v)$ to dominate $V_i$.

Let $H = (V,F)$ be a graph with no $TK_h$ and recall that $K,\alpha,l,q$ are given in (1). It will be convenient to work in the double-cover of $H$ which we are going to define next. We say that the bipartite graph $G = (V,V',E)$ is associated with $H$ if $V' = \{v' : v \in V\}$ and we have $vu' \in E$ if and only if $vu \in F$ or $u = v$. In other words, edge $vu \in F$ corresponds to two edges $vu'$, $u'v$ in $E$ and $vv' \in E$ for every $v$ from $V$. Let $\gamma'(G)$ denote the minimum size of a set $S \subseteq V$ which dominates $V'$ in $G$. Before discussing the first phase of the algorithm, we will mention a few facts on the relation between $H$ and $G$.

\begin{itemize}
  \item \textbf{Fact 10.} $X$ is a dominating set in $H$ if and only if $N_G(X) = V'$.
\end{itemize}

\textbf{Proof.} Suppose $X$ is a dominating set in $H$. Then every vertex $u \in V(H) \setminus X$ is adjacent to a vertex $v \in X$, and so $u'v \in E(G)$, and for every $u \in X$, $uu' \in E(G)$. Now, let $Y \subset V$ and let $u \in V(H) \setminus Y$. Since $N_G(Y) = V'$, there is a vertex $v \in Y$ such that $u'v \in E(G)$. Then $uv \in E(H)$ as $u' \neq v'$.

In particular, we have

$$\gamma(H) = \gamma'(G).$$

Rather than studying topological minors in $G$ in relation to topological minors in $H$, we note the following simple fact.

\begin{itemize}
  \item \textbf{\textbullet} Fix $h \geq 1$.
\end{itemize}
Distributed Approximation for MDS

Fact 11. Let \( D \subseteq V \), \( v \in V \setminus D \), and suppose there is a \( v, D \)-fan in \( G \) of size \( 2t - 1 \) such that every path has length two. Then \( v \in D_{t,2} \) in \( H \).

Proof. A \( v, D \)-fan in \( G \) consists of paths of the form \( v, u', w \), where \( w \in D \). Such paths are mapped to paths of length one (if the corresponding copy of \( u' \) is in \( D \)) or paths of length two (if the corresponding copy of \( u' \) is not in \( D \)). Thus for every vertex from \( D \) there are at most two paths that are mapped to ones that contain this vertex in \( H \). As a result we can always choose \( t \) vertex disjoint paths of a \( v, D \)-fan in \( G \) out of the \( 2t - 1 \) paths of \( v, D \)-fan in \( H \).

Lemma 3 and Fact 11 give the following corollary.

Corollary 12. If \( H \) has no \( TK_h \) and \( D \subseteq V \) is such that \( N_G(D) = V' \), then the number of vertices \( v \in V \) such that there is a \( v, D \)-fan in \( G \) of size \( 2h - 1 \) such that each path has length two is \( O(|D|) \).

Since we will partition sets \( N_G(v) \subseteq V' \) (for \( v \in V \)), all the partition classes will be subsets of \( V' \). It will be convenient to introduce the following notion.

Definition 13. We say that the partition \( \{V'_0, V'_1, \ldots, V'_s\} \) of \( N_G(v) \) from Fact 9 is associated with \( v \).

Let \( D^* \subseteq V \) be an optimal set which dominates \( V' \) in \( G \) and let \( V^* \) be the set of vertices \( v \in V \setminus D^* \) such that there is a \( v, D^* \)-fan consisting of \( K + 1 \) paths, each of length at most two. Then, by Corollary 12, \( |V^*| = O(|D^*|) \). In view of the previous discussion adding vertices from \( V^* \cup D^* \) to our solution results in a dominating set of size \( O(|D^*|) \). The remaining vertices \( v \) from \( V \setminus (V^* \cup D^*) \) will have \( N(v) \) dominated by a “few” vertices from \( D^* \). Suppose \( v \in V \setminus (V^* \cup D^*) \). Then \( N(v) \) is dominated by vertices from \( D^* \) and so there exist \( d_1, \ldots, d_m \in D^* \) for some \( m \leq K \), such that \( N(v) \subseteq \bigcup N(d_i) \). Therefore \( \{d_1, \ldots, d_m\} \) is a cover of \( N(v) \) and by Fact 6, \( \{d_1, \ldots, d_m\} \) gives an \((a,q,l,K)\)-pseudo-cover which belongs to \( T(v) \). In addition, by Fact 9, if \( \{V'_0, V'_1, \ldots, V'_s\} \) is the partition associated with \( v \), then for every \( i \geq 1 \), \( V'_i \subseteq N(d_j) \) for some \( j \leq m \). In view of the previous discussion, we have the following fact.

Fact 14. Let \( D^* \subseteq V \) be an optimal set which dominates \( V' \) in \( G \) and let \( v \in V \setminus (D^* \cup V^*) \). In addition, let \( V'_0, V'_1, \ldots, V'_s \) be the partition associated with \( v \), and for \( i \geq 1 \), let \( U_i = \{x \in S : S \in T(v) \land V'_i \subseteq N(x)\} \). Then \( U_i \setminus D^* \neq \emptyset \).

In the main part of our algorithm, we will find a set \( D \subseteq V \) which dominates some vertices from \( V' \) so that when \( N_G(D) \) is removed from \( V' \), then every vertex in \( V' \) has its degree bounded by a constant. To extend \( D \) and find a set which dominates all vertices from \( V' \) we will rely on the following simple observation.

Fact 15. Let \( s \in Z^+ \) and let \( X = (V, V', E) \) be a bipartite graph such that for every vertex \( v \in V \), \( d(v) \leq s \) and for every \( v' \in V' \), \( d(v') \geq 1 \). Then \( \gamma'(X) \geq |V'|/s \).

Proof. If \( D \subseteq V \) is an optimal set which dominates \( V' \), then \( s|D| \geq |E(D,V')| \geq |V'| \). Recall that for every \( v \in V \), \( T(v) \neq \emptyset \) because \((v) \in T(v) \). To motivate our discussion suppose first that \( |T(v)| = 1 \). Then either \( v \in V^* \) (defined above) or otherwise, in view of Fact 14, \( v \in D^* \). In either case we can add such a vertex to our solution. In fact a stronger observation is true, if for some \( V_i \) in the partition associated with \( v \), \( v \) is the only vertex in some \( S \in T(v) \) such that \( V_i \subseteq N(v) \), then \( v \in V^* \cup D^* \). Unfortunately, in many cases there
will be more than one vertex $u$ such that $V_i \subseteq N(u)$ and the challenge is to select one which will lead to a constant approximation of $\gamma'(G)$. The assumption that $H$ has no $TK_h$ implies that the number of choices of $u$ is bounded by a constant which depends on $h$ only but only some of these vertices will be good choices. The algorithm will consist of two main phases. The first one deals with those vertices $v$ for which $|\mathcal{T}(v)| = 1$, and the second one addresses the more difficult case.

**Phase 1**

Input: Graph $H = (V, E)$ with no $TK_h$.

1. Consider the double cover $\mathcal{C} = (V, V', E)$ of $H$. Let $D_1 := \emptyset$.
2. Compute $\mathcal{T}(v)$ for every $v \in V$. If $|\mathcal{T}(v)| \geq 2$ then mark $v$. Add all unmarked vertices to $D_1$ and delete all vertices from $V'$ dominated by $D_1$.

Next fact follows from previous discussion.

► **Lemma 16.** Let $D_1$ be the set obtained in Phase 1. Then $|D_1| = \Theta(\gamma(H))$.

We will now continue our analysis assuming we have set $D_1$ obtained in Phase 1 and $G$ has been modified by possibly deleting some vertices from $V'$. Let $V''$ denote the remaining vertices in $V'$, that is $V'' := V' \setminus N_G(D_1)$. In addition, we shall use $V_i''$ to denote $V_i \cap V''$.

Consider a sequence of constants $M_0,M_1,\ldots,M_h$ such that $M_h \geq \binom{h}{2} + h$ and for every $1 \leq i \leq h$, we have

$$M_{i-1} > (2^{(K+1)C} + 1)(Cq + M_i 2^{(K+1)C}),$$

where $C = 2\left(\frac{\binom{h}{2}}{\alpha}\right)^K$ and $K, q, s, t$ are defined in (1). In fact, we will only need that $M_{h-1} \geq \binom{h}{2} + h$ but in the process described below, which uses constants $M_i$, we will allow it to continue more than $h - 1$ times. Let $v \in V \setminus D_1$ and let $V_i''$ be a set in the partition associated with $v$ which satisfies

$$|V_i''| \geq M_0.$$ 

We set $v_0 := v$ and consider $V_i'''$. For every $v_1 \in V \setminus (D_1 \cup \{v\})$ which belongs to some $S \in \mathcal{T}(v)$ and is such that $V_i''' \subseteq N(v_0)$ take partition $W_0'', W_1'', \ldots, W_h''$ associated with $v_1$ and let $W_i''' := W_i'' \cap V''$. We have $p \leq 2^{(K+1)C}$ by Fact 8, $|W_i''| \leq Cq$ by Fact 9. Let $\mathcal{P}_{v_1,v_0}^{V_i'''} = \{W_i'' \cap V_i''' : |W_j'' \cap V_j'''| \geq M_1 \land j \geq 1\}$. We have $\bigcup \mathcal{P}_{v_1,v_0}^{V_i'''} \subseteq V_i'''$ and $\bigcup_{j \geq 0} V_i'''' \cap W_j'' = V_i'''$. Therefore,

$$|\bigcup \mathcal{P}_{v_1,v_0}^{V_i'''}| \geq |V_i'''| - Cq - 2^{(K+1)C} \cdot M_1.$$

We call sets $W_j'' \cap V_i''' \in \mathcal{P}_{v_1,v_0}^{V_i'''}$ fragments. Now we iterate the process for every fragment $W_j'' \cap V_i''' \in \mathcal{P}_{v_1,v_0}^{V_i'''}$, that is, we consider $v_2 \in V \setminus (D_1 \cup \{v_1\})$ such that for some $S \in \mathcal{T}(v_1)$ we have $v_2 \in S$, $W_j'' \subseteq N_G(v_2)$ and $v_2 \neq v_i$ for $i < 2$. Define $\mathcal{P}_{v_1,v_2,v_0}^{W_j'' \cap V_i''' = \{Z_k'' \cap W_j'' \cap V_i''' : |Z_k'' \cap W_j'' \cap V_i'''| \geq M_2 \land k \geq 1\}}$. We have

$$|\bigcup \mathcal{P}_{v_1,v_2,v_0}^{W_j'' \cap V_i'''| \geq |W_j'' \cap V_i'''| - 2Cq - 2^{(K+1)C} \cdot M_2.$$ 

We repeat the process for as long as possible (See Figure 3 for an illustration). We will now establish three claims about the above process. First claim states that the process must end after $h - 1$ steps or the original graph $H$ contains a $TK_h$. The second claim shows that in
the sequences of vertices obtained by the process, the last vertices are a good choice to be added to a solution. Finally, the last claim states that when last vertices are added then all but a constant number of vertices from $V''_i$ are dominated.

▶ Claim 17. If $v_0, v_1, \ldots, v_i$ is a sequence obtained in the above process, then $i \leq h - 2$.

Proof. Suppose $i \geq h - 1$. Then $\{v_0, \ldots, v_{h-1}\}$ contains distinct vertices, every fragment $U \in \mathcal{P}^W_{v_0, \ldots, v_{h-1}}$ has size $|U| \geq M_{h-1}$, and $U \subseteq N(v_0) \cap \cdots \cap N(v_{h-1})$. Since $M_{h-1} \geq \binom{h}{2} + h$, $G$ contains $K_{h, \binom{h}{2} + h}$, and as a result $H$ contains $TK_h$. ◀

For every maximal sequence $v_0, v_1, \ldots, v_j$ obtained in the process above for $V''_i$, we add $v_j$ to $D(V''_i)$.

Let $Z$ be the set of vertices $z \in V$ that belong to some $S$ where $S \in \mathcal{T}^{(\leq h)}(d)$ for some $d \in D^*$. Then we have $|Z| = O(|D^*|)$ because there is a constant number of vertices which belong to some $S \in \mathcal{T}^{(\leq h)}(d)$. In addition, we have the following.

▶ Claim 18. We have $D_2(V''_i) \subseteq V^* \cup D^* \cup Z$.

Proof. Suppose $v_0, \ldots , v_i$ is a maximal sequence. Let $U$ be a fragment in $\mathcal{P}^W_{v_0, \ldots, v_i}$ and let $X'_0, \ldots, X'_j$ denote the partition associated with $v_i$. Then $U \subseteq X'_j$ for some $j \geq 1$. By Fact 14, either $v_i \in V^* \cup D^*$ or for at least one $d \in D^*$, we have $d$ in some $S \in \mathcal{T}(v_i)$ and $X'_j \subseteq N_G(d)$. Recall that $\mathcal{T}(d)$ is non-empty and so there is a partition associated with $d$, but it can be trivial. We have $|U \cap Y'_j| \geq M_{i+1}$ for at least one set $Y'_j$ such that $j \geq 1$ and $Y'_j$ is in the partition associated with $d$. Thus $d$ is an option for $v_{i+1}$ and so, since the sequence is maximal $d = v_k$ for some $k < i$. We have $i \leq h - 2$ by Claim 17. Consequently $v_i \in V^* \cup D^* \cup Z$.

Finally we show that vertices from $D_2(V''_i)$ cover all but a constant number of vertices in $V''_i$. 

![Figure 3](image-url) Constructing fragments from a set $U$ as vertices are added to a sequence starting with $v_0$. 

We can now describe the second phase of the algorithm.

To finish our analysis, we will show that

\[ A. \text{Czygrinow, M. \Hanckowiak, W. \Wawrzyniak, and M. \Witkowski} \]
Proof. Let $D^* \subseteq V$ be such that $|D^*| = \gamma^*(G)$. From Lemma 16, $|D_1| = O(\gamma(H))$. From Claim 18, we have $D_2(V_i') \subseteq V^* \cup D^* \cup Z$ for every vertex $v$ and every set $V_i'$ considered in step 3. Thus $D_2 \subseteq V^* \cup D^* \cup Z$ and since $|Z| = O(\gamma(H))$, we have $|D_2| = O(\gamma(H))$. Let $X := G[V \setminus (D_1 \cup D_2), V'' \cup \bigcup_{v \in D_2} N(v)]$ and let $v \in V \setminus (D_1 \cup D_2)$. By Claim 19, for every set $V_i'$ with $i \geq 1$ in the partition associated with $v$, only a constant number of vertices $L$ are not dominated by vertices in $D_2$. Since, by Fact 8 the number of sets $V_i'$ is at most $2^{(K+1)C}$ and by Fact 9, $|V_0'| \leq Cq$, we have $d_X(v)$ bounded by some constant $p$ for every $v \in V \setminus (D_1 \cup D_2)$. By Fact 15, $\gamma'(X) \geq |V'' \cup \bigcup_{v \in D_2} N(v)|/p$ and at the same time vertices in $V'' \cup \bigcup_{v \in D_2} N(v)$ can only be dominated by vertices in $V \setminus (D_1 \cup D_2)$ and so $\gamma'(X) \leq |D^*|$. Consequently, $|D_3| = |V'' \cup \bigcup_{v \in D_2} N(v)| = O(\gamma'(X)) = O(|D^*|)$.  

Proof of Theorem 1. Combining Fact 20, Fact 21 and Fact 22 shows that given a graph $H$ with no $TK_h$, Algorithm DominatingSet finds in $L_h$ rounds a dominating set $D$ such that $|D| \leq C_h \gamma(H)$ for some constants $L_h$ and $C_h$ which depend on $h$ only.

As noted in the introduction, Theorem 1 in connection with methods developed in [2] (Theorem 3.4) immediately imply Theorem 2.

References

Abstract

We study the oritatami model for molecular co-transcriptional folding. In oritatami systems, the transcript (the “molecule”) folds as it is synthesized (transcribed), according to a local energy optimisation process, which is similar to how actual biomolecules such as RNA fold into complex shapes and functions as they are transcribed. We prove that there is an oritatami system embedding universal computation in the folding process itself.

Our result relies on the development of a generic toolbox, which is easily reusable for future work to design complex functions in oritatami systems. We develop “low-level” tools that allow to easily spread apart the encoding of different “functions” in the transcript, even if they are required to be applied at the same geometrical location in the folding. We build upon these low-level tools, a programming framework with increasing levels of abstraction, from encoding of instructions into the transcript to logical analysis. This framework is similar to the hardware-to-algorithm levels of abstractions in standard algorithm theory. These various levels of abstractions allow to separate the proof of correctness of the global behavior of our system, from the proof of correctness of its implementation. Thanks to this framework, we were able to computerise the proof of correctness of its implementation and produce certificates, in the form of a relatively small number of proof trees, compact and easily readable/checkable by human, while encapsulating huge case enumerations. We believe this particular type of certificates can be generalised to other discrete dynamical systems, where proofs involve large case enumerations as well.

2012 ACM Subject Classification Theory of computation → Models of computation, Applied computing → Life and medical sciences, Hardware → Biology-related information processing

Keywords and phrases Molecular computing, Turing universality, co-transcriptional folding

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.23

Related Version A full version of the paper is available at [7], https://arxiv.org/abs/1508.00510.
1 Introduction

Oritatami model was introduced in [6] to try to understand the kinetics of co-transcriptional folding. This process has been shown to play an important role in the final shape of biomolecules [1], especially in the case of RNA [4]. The rationale of this choice is that the wetlab version of Oritatami already exists, and has been successfully used to engineer shapes with RNA in the wetlab [8].

In oritatami, we consider a finite set of bead types, and a periodic sequence of beads, each of a specific bead type. Beads are attracted to each other according to a fixed symmetric relation, and in any folding (a folding is also called a configuration), whenever two beads attracted to each other are found at adjacent positions, a bond is formed between them.

At each step, the latest few beads in the sequence are allowed to explore all possible positions, and we keep only those positions that minimise the energy, or otherwise put, those positions that maximise the number of bonds in the folding. “Beads” are a metaphor for domains, i.e. subsequences, in RNA and DNA.

Previous work on oritatami includes the implementation of a binary counter [6], the Heighway dragon fractal [12], folding of shapes at small scale [3], and NP-hardness of the rule minimization [15, 9] and of the equivalence of non-deterministic oritatami systems [10].

Main result. In this paper, we construct a “universal” set of 542 bead types, along with a single universal attraction rule for these bead types, with which we can simulate any tag system, and therefore any Turing machine \( \mathcal{M} \), within a polynomial factor of the running time \( \mathcal{M} \). The reduction proceeds as follows:

\[
\text{Turing machine} \xrightarrow{[16, 13]} \text{Cyclic tag system} \xrightarrow{\text{Prop. 2}} \text{Skipping cyclic tag system} \xrightarrow{\text{Lem. 3}} \text{Oritatami system}
\]

Our result relies on the development of a generic toolbox, which is easily reusable for future work to design complex functions in oritatami systems.

Proving our designs. The main challenge we faced in this paper was the size of our constructions: indeed, while we developed higher-level geometric constructs to program useful shapes, there is a large number of possible interactions between all different parts of the sequence. Getting solid proofs on large objects is a common problem in discrete dynamical systems, for instance on cellular automatons [5, 2] or tile assembly systems [11]. In this paper, we introduce a general framework to deal with that complexity, and prove our constructions rigorously. This method proceeds by decomposing the sequence into different modules, and the space into different areas: blocks, where exactly one step of the simulation is performed, which are composed of bricks, where exactly one module grows. We can then reason on the modules separately, and only deal with interactions at the border between all possible modules that can have a common border.

2 Definitions and Main results

2.1 Oritatami Systems

Let \( B \) be a finite set of bead types. A configuration \( c \) of a bead type sequence \( p \in B^+ \cup B^0 \) is a directed self-avoiding path in the triangular lattice \( \mathbb{T} \), where for all integer \( i \), vertex \( c_i \) of \( c \) is defined as \( \mathbb{T} = (\mathbb{Z}^2, \sim) \), where \((x, y) \sim (u, v)\) if and only if \((u, v) \in \bigcup_{\epsilon = \pm 1}\{(x + \epsilon, y), (x, y + \epsilon), (x + \epsilon, y + \epsilon)\}\). Every position \((x, y)\) in \( \mathbb{T} \) is mapped in the euclidean plane to \( x \cdot \hat{E} + y \cdot \hat{SW} \) using the vector basis \( \hat{E} = (1, 0) \) and \( \hat{SW} = \text{RotateClockwise}(\hat{E}, 120^\circ) = \left(-\frac{1}{2}, -\frac{\sqrt{3}}{2}\right) \).

\footnote{The triangular lattice is defined as \( \mathbb{T} = (\mathbb{Z}^2, \sim) \), where \((x, y) \sim (u, v)\) if and only if \((u, v) \in \bigcup_{\epsilon = \pm 1}\{(x + \epsilon, y), (x, y + \epsilon), (x + \epsilon, y + \epsilon)\}\). Every position \((x, y)\) in \( \mathbb{T} \) is mapped in the euclidean plane to \( x \cdot \hat{E} + y \cdot \hat{SW} \) using the vector basis \( \hat{E} = (1, 0) \) and \( \hat{SW} = \text{RotateClockwise}(\hat{E}, 120^\circ) = \left(-\frac{1}{2}, -\frac{\sqrt{3}}{2}\right) \).}
is labelled by \( p_i \). \( c_i \) is the position in \( T \) of the \((i + 1)\)th bead, of type \( p_i \), in configuration \( c \). A partial configuration of a sequence \( p \) is a configuration of a prefix of \( p \).

For any partial configuration \( c \) of some sequence \( p \), an elongation of \( c \) by \( k \) beads (or \( k\)-elongation) is a partial configuration of \( p \) of length \(|c| + k\) extending by \( k \) positions the self-avoiding path of \( c \). We denote by \( C_p \) the set of all partial configurations of \( p \) (the index \( p \) will be omitted when the context is clear). We denote by \( c^p\) the set of all \( k\)-elongations of a partial configuration \( c \) of sequence \( p \).

**Oritatami systems.** An oritatami system \( O = (p, \heartsuit, \delta) \) is composed of (1) a (possibly infinite) bead type sequence \( p \), called the transcript, (2) an attraction rule, which is a symmetric relation \( \heartsuit \subseteq B^2 \), and (3) a parameter \( \delta \) called the delay. \( O \) is said periodic if \( p \) is infinite and periodic. Periodicity ensures that the “program” \( p \) embedded in the oritatami system is finite (does not harcode any specific behavior) and at the same time allows arbitrary long computation.

We say that two bead types \( a \) and \( b \) attract each other when \( a \heartsuit b \). Furthermore, given a (partial) configuration \( c \) of a bead type sequence \( q \), we say that there is a bond between two adjacent positions \( c_i \) and \( c_j \) of \( c \) in \( T \) if \( q_i \heartsuit q_j \) and \(|i - j| > 1 \). The number of bonds of configuration \( c \) of \( q \) is denoted by \( H(c) = |\{(i, j) : c_i \sim c_j, j > i + 1, \text{ and } q_i \heartsuit q_j\}| \).

**Oritatami dynamics.** The folding of an oritatami system is controlled by the delay \( \delta \). Informally, the configuration grows from a seed configuration (the input), one bead at a time. This new bead adopts the position(s) that maximise(s) the potential number of bonds the configuration can make when elongated by \( \delta \) beads in total. This dynamics is oblivious as it keeps no memory of the previously preferred positions; it differs thus slightly from the hasty dynamics studied in [6]; it might also be considered as closer to experimental conditions such as in [8].

Formally, given an oritatami system \( O = (p, \heartsuit, \delta) \) and a seed configuration \( \sigma \) of a seed bead type sequence \( s \), we denote by \( C_{\sigma, p} \) the set of all partial configurations of the sequence \( s \cdot p \) elongating the seed configuration \( \sigma \). The considered dynamics \( \mathcal{D} : 2^{C_{\sigma, p}} \rightarrow 2^{C_{\sigma, p}} \) maps every subset \( S \) of partial configurations of length \( \ell \) elongating \( \sigma \) of the sequence \( s \cdot p \) to the subset \( \mathcal{D}(S) \) of partial configurations of length \( \ell + 1 \) of \( s \cdot p \) as follows:

\[
\mathcal{D}(S) = \bigcup_{c \in S} \arg \max_{c \in \sigma^1} \left( \max_{\eta \in \gamma^{(\delta - 1)}} H(\eta) \right)
\]

The possible configurations at time \( t \) of the oritatami system \( O \) are the elongations of the seed configuration \( \sigma \) by \( t \) beads in the set \( \mathcal{D}^t(\{\sigma\}) \).

We say that the oritatami system is deterministic if at all time \( t \), \( \mathcal{D}^t(\{\sigma\}) \) is either a singleton or the empty set. In this case, we denote by \( c^t \) the configuration at time \( t \), such that: \( c^0 = \sigma \) and \( \mathcal{D}^t(\{\sigma\}) = \{c^t\} \) for all \( t > 0 \); we say that the partial configuration \( c^t \) folds (co-transcriptionally) into the partial configuration \( c^{t+1} \) deterministically. In this case, at time \( t \), the \((t + 1)\)th bead of \( p \) is placed in \( c^{t+1} \) at the position that maximises the number of bonds that can be made in a \( \delta \)-elongation of \( c^t \).

We say that the oritatami system halts at time \( t \) if \( t \) is the first time for which \( \mathcal{D}^t(\{\sigma\}) = \varnothing \). The folding process may only stop because of a geometric obstruction (no more elongation is possible because the configuration is trapped in a closed area).

Please refer to Fig. 1(d) and 1(e) for examples of the dynamical folding of a transcript. Observe that a given transcript may fold (deterministically) into different paths because of its interactions with its local environment (see section 2.3 for more details).
2.2 Main result

Our main result consists in proving the following theorem that demonstrates that oritatami systems are able to complete arbitrary Turing computation. It shows in particular that deciding whether a given oritatami system folds into a finite size shape for a given seed is undecidable.

▶ Theorem 1 (Main result). There is a fixed set $B$ of 542 bead types with a fixed attraction rule $\heartsuit$ on $B$, together with two encodings:
- $\pi$ that maps in polynomial time, any single tape Turing machine $M$ to a bead type sequence $\pi_M \in B^*$;
- $(s, \sigma)$ that maps in polynomial-time, any single-tape Turing machine $M$ and any input $x$ of $M$ to a seed configuration $\sigma_M(x)$ of a bead type sequence $s_M(x)$ of length $O_M(|x|)$, linear in the size of the input $x$ (and polynomial in $|M|$);

such that: For any single tape Turing machine $M$ and every input $x$ of $M$, the deterministic and periodic oritatami system $O_M = ((\pi_M)^\infty, \heartsuit, 3)$ whose transcript has period $\pi_M$ and whose delay is $\delta = 3$, halts its folding from the seed configuration $\sigma_M(x)$ if and only if $M$ halts on input $x$. Furthermore, for all $t$ and all input $x$ of $M$, if $M$ halts on $x$ after $t$ steps, then the folding of $O_M$ from seed configuration $\sigma_M(x)$ halts after folding $O_M(t^4 \log^2 t)$ beads.

There is one Turing-universal periodic transcript. Note that if we apply this theorem to an intrinsically universal single tape Turing machine $U$ (see [14]), then we obtain one single absolutely fixed transcript $\pi_U$ such that the deterministic and periodic oritatami system $O_U = ((\pi_U)^\infty, \heartsuit, 3)$ with 542 bead types can simulate efficiently the halting of any Turing machine $M$ on any input $x$ using a suitable seed configuration obtained via the encoding of $M$ and $x$ in $U$. It follows that this absolutely fixed oritatami system consisting of one single periodic transcript is able of arbitrary Turing computation.

From now on, we only consider deterministic periodic oritatami systems with delay $\delta = 3$.

2.3 Basic design tool: Glider/Switchback

As a warm-up, let us introduce a special type of bead sequence (see Fig. 1) that, depending on the initial context of its folding, either folds as a glider (a long and thin self-supported shape heading in a fixed direction) or as switchbacks (a narrow and high shape allowing compact storage). This only requires a small number of distinct beads types (12 per switchbacks, that can be repeated every 4 switchbacks). This is achieved by designing a rule with minimum interactions ensuring minimum interferences between both folding patterns. Compatibility between the glider and the turns in switchbacks is ensured by aligning the switchback turns with the turns of the glider, exploiting thus the similarity of their finger-like shape there.

This glider/switchback sequence will be used to store (as switchbacks) and expose (as glider) specific information encoded in the transcript when needed.

2.4 Skipping Cyclic Tag Systems and Turing-Universality

Our proof of the Turing-universality of oritatami systems consists in simulating a special kind of cyclic tag systems (CTS), called skipping cyclic tag system. Cook introduced CTS in [2] and proved that they combined the tremendous advantage of simulating efficiently any Turing machines, while not requiring a random access lookup table, which makes simulation a lot easier.
A skipping cyclic tag system (SCTS) consists of a cyclic list of \( n \) words \( \alpha = \langle \alpha^0, \ldots, \alpha^{n-1} \rangle \in \{0,1\}^* \), called appendants, and an initial dataword \( u^0 \in \{0,1\}^* \). Intuitively, \( \alpha \) encodes the program and \( u^0 \) encodes the input. Its configuration at time \( t \) consists of a marker \( m^t \), recording the index of the current appendant at time \( t \), and a dataword \( u^t \). Initially, \( m^0 = 0 \) and the dataword is \( u^0 \). At each time step \( t \), the SCTS acts deterministically on configuration \( (m^t, u^t) \) in one of three ways:

(Halt step) If \( u^t \) is the empty word \( \epsilon \), then the SCTS halts;\(^4\)

(Nop step) If the first letter \( u^0_0 \) of \( u^t \) is 0, then \( u^0_0 \) is deleted and the marker moves to the next appendant cyclically; i.e., \( m^{t+1} = (m^t + 1) \mod n \) and \( u^{t+1} = u^t_{1} \cdots u^t_{|u^t| - 1} \).

(Skip-append step) If \( u^0_0 = 1 \), then \( u^0_0 \) is deleted, the next appendant \( \alpha^{(m^t+1) \mod n} \) is appended onto the right end of \( u^t \), and the marker moves to the second next appendant: i.e., \( u^{t+1} = u^t_{1} \cdots u^t_{|u^t| - 1} \cdot \alpha^{(m^t+1) \mod n} \) and \( m^{t+1} = (m^t + 2) \mod n \).

For example, consider the SCTS \( E = ((110, \epsilon, 11, 0); u^0 = 010) \). Its execution \( ([m^t; u^t]) \) is:

\[
\]

\(^4\) Note that SCTS halting condition requires the dataword to be empty as opposed to \([2, 16]\) where the computation of a cyclic tag system is said to end also if it repeats a configuration.
**Turing universality.** A sequence of articles and thesis by Cook [2], and Neary and Woods [16, 13], allows to show that SCTS are able to simulate any Turing machine efficiently in the following sense: (proof omitted see [7])

▶ **Proposition 2** ([16, 13]). Let $M$ be a deterministic Turing machine using a single tape. There is a polynomial algorithm that computes a skipping cyclic tag system $S_M$, together with a linear-time encoding $u_M(x)$ of the input $x$ of $M$ as an input dataword for $S_M$, such that for all input $x$: $S_M$ halts on input dataword $u_M(x)$ if and only if $M$ halts on input $x$. Furthermore, for all $t$, if $M$ halts after $t$ steps, then $S$ halts after $O_M(t^2 \log t)$ steps. Moreover, the number of appendants of $S$ is a multiple of 4.

In order to prove Theorem 1, we are thus left with proving that there is an oritatami system that simulates in quadratic time any SCTS system (see Theorem 6 in [7] for a precise statement).

### 3 The block simulation of SCTS: Proving the correctness of local folding is enough

Given a SCTS $S$, we design an oritatami system $O_S$ that folds into a version, at a larger scale, of the annotated trimmed space-time diagram of $S$ (or trimmed diagram for short) defined as follows:

**Trimmed diagram of SCTS.** Any SCTS proceeds as follows: it trims all the leading 0s in the dataword and then appends the currently marked appendant when it reads the first 1 (if any; otherwise it halts). It is thus natural to group all these steps (trim leading 0s and process the leading 1) as one single macro step. This motivates the following representation. Given a SCTS $(\alpha^0, \ldots, \alpha^{n-1}; u^0)$, we denote by $0 \leq t_1 < t_2 < \cdots$ all the times $t$ such that the dataword $u^t$ starts with letter 1 and set $t_0 = -1$ by convention. Let us now group all deletion steps occurring during steps $t_i + 1$ to $t_{i+1} - 1$ by simply indicating in exponent the marker $m^t$ before each letter read. In the case of our STCS $E$, we have $t_0 = -1, t_1 = 1, t_2 = 3, t_3 = 4$ and its execution is now represented as: $[0]_0 [1]_1 0 \xrightarrow{[2:11]} [3]_0 [0]_1 1 \xrightarrow{[1:2]} [2]_1 \xrightarrow{[k:0]} [0]_0 [1]_1 \text{Halt}$.

Now, let’s align the resulting datawords in a 2D diagram according to their common parts:

```
  t_0 \downarrow  t_1 \downarrow  t_2 \downarrow  t_3 \downarrow
  [0]_0 [1]_1 0 \xrightarrow{[2:11]} [3]_0 [0]_1 1 \xrightarrow{[1:2]} [2]_1 \xrightarrow{[k:0]} [0]_0 [1]_1 \text{Halt}
```

This defines the annotated trimmed space-time diagram for the SCTS $E$. We refer to Lemma 4 in [7] for the formal definition for an arbitrary SCTS.

**The transcript.** The proof of Theorem 1 (see Theorem 6 in [7]) relies on constructing a transcript (and a fixed rule) that will reproduce faithfully the trimmed diagram of the simulated STCS. Figure 2 illustrates the folded configuration of the transcript corresponding to SCTS $E$. Macroscopically, the transcript folds into a zig-zag sequence of blocks, each performing a specific operation.

The current dataword is encoded at the bottom of each row of blocks: 0s are encoded by a spike, and 1s are encoded by a flat surface.
The seed configuration encodes the initial dataword and opens the first zig row at which the folding of the transcript starts. Letters 0 and 1 are encoded by a spike (see Fig. 3(a)) and a flat surface (see Fig. 3(b)) respectively.

In each zig row (left to right), the transcript folds into a series of Read\(\triangleright\) blocks (trimming the leading 0s from the dataword encoded above), and then into a Read\(\triangleright\) block, if the dataword contains a 1, or into a Halt block terminating the folding, otherwise; this is the zig-up phase. Then, the transcript starts the zig-down phase which consists in folding into Copy\(\triangleright\) block copying the remaining letters of the dataword encoded above to the bottom of the row; once the end of the dataword is reached, the transcript folds into an Append\(\cdot\)Return block which encodes, at the bottom of the row, the currently marked appendant, and finally, opens the next zag row.

In each zag row (right to left), the transcript folds into Copy\(\triangleright\) blocks copying the new dataword encoded above to the bottom of the row. For the leftmost letter, the transcript folds into the special Copy\(\triangleright\)LineFeed block which also opens the next zig row.

The transcript is a periodic sequence whose period is the concatenation of \(n\) bead type sequences \([\text{Appendant } \alpha^0], \ldots, \text{Appendant } \alpha^{n-1}\) called segments, each encoding one appendant.

Encoding of the marker. Read\(\triangleright\) and Append\(\cdot\)Return blocks consist of the folding of exactly one segment, whereas Copy\(\triangleright\), Copy\(\triangleright\) and Copy\(\triangleright\)LineFeed consist of the folding of exactly \(n\) segments. It follows that the appendant encoded in the leading segment folded inside each block corresponds to the currently marked appendant in the simulated SCTS. As a consequence, the appendant contained in the folded Append\(\cdot\)Return block is indeed the appendant to be appended to the dataword.

The segment sequence. Each segment \([\text{Appendant }\alpha^i]\) encodes the appendant \(\alpha^i\) as a sequence of \(6 + |\alpha^i|\) modules: one of each module \(A, B,\) and \(C\), then \(|\alpha^i|\) of module \(D\), then one of each module \(E, F\) and \(G\). Each module is a bead type sequence that plays a particular role in the design:

- **Module** \(A\) folds into the initial scaffold upon which the next modules rely.
- **Module** \(B\) detects if the dataword is empty: if so, it folds to the left so as the folding gets trapped in a closed space and halts; otherwise, it folds to the right and the folding continues.
- **Module** \(C\) detects the end of the dataword and triggers the appending of the marked appendant accordingly.
- **Module** \(D\) encodes each letter of the appendant: its two variants \(D_0\) and \(D_1\) encode respectively 0s and 1s.
- **Module** \(E\) ensures by padding that all appendant sequences have the same length when folded (even if the appendant have different length). It serves two other purposes: Module \(B\) senses its presence to detect if the dataword is empty; and its folding initiates the opening of the zag row once the marked appendant has been appended to the dataword.
- **Module** \(F\) is the scaffold upon which module \(G\) folds. It is specially designed to induce two very distinct shapes on \(G\) depending on the initial shift of \(G\). Furthermore, when module \(F\) is exposed, module \(C\) folds along \(F\) which triggers the appending of the marked appendant encoded by the modules \(D\) following \(C\).
- **Module** \(G\) is the “logical unit” of the transcript. It implements three distinct functions which are triggered by its interactions with its environment: (1) reading the \(0^* (1 | \epsilon)\) prefix of the dataword, (2) copying a letter of the dataword, and (3) opening the next zig row at the leftmost end of each zag row.
(a) Folding of the oritatami system simulating the STCS $E$.

(b) Exploded view of the bricks and modules inside the blocks involved in the simulation above.

**Figure 2** Folding of the transcript simulating the STCS $E$, and some block internal structures.

We call *bricks* the folding of each of these modules. The blocks into which the transcript folds, depend on the bricks in which its modules fold, as illustrated in Fig. 2(b). We refer to sections C to F in [7] for the description of blocks in terms of bricks and of how they articulate with each other to produce the desired macroscopic folding pattern.

The full description of each of these sequences is given in Section F in [7].

Let $S = (\alpha^0, \alpha^1, \ldots, \alpha^{n-1}; u^0)$ be a SCTS, and, as before, let for all integer $i \geq 0$, $t_i$ be the $i$th step where $u^i$ starts with 1 (starting from 0, i.e. $t_0$ is the first step where $u^0$ starts with 1). The following lemma shows that the transcript described above folds indeed into blocks that simulate the trimmed diagram of $S$. Proposition 2 and Theorem 1 are direct corollaries of this lemma.
Lemma 3 (Key lemma). There is a bead set type $B$ and a rule such that: for every SCTS $S$, there are $\pi_S$ and $(\sigma_S, s_S)$ defined as in Theorem 1 such that, for every initial dataword $u^0$, the (possibly infinite) final folded path of the oritatami system $O_S = ((\pi_S)^\infty, (\sigma_S), \delta = 3)$ from the seed configuration $s(u^0)$ is exactly structured as the following sequence of blocks organized in zig and zag rows as follows: (recall Fig. 2(a))

- First, the block Seed($u^0$) ending at coordinates $(-1, 0)$.
- Then, for $i \geq 0$, the $i$-th row consists of a zig row located between $y = 2(i - 1)h + 1$ and $y = 2ih$, and a zag row located between $y = 2ih + 1$ and $y = 2(i + 1)h$, composed as follows:
  - (Compute) if $u^{1+t_i} = 0^*1 \cdot s$ and if $s \neq \epsilon$ or $\alpha^{1+i+t_{i+1}} \neq \epsilon$: then $r = t_{i+1} - t_i - 1$ and:
    - the zig-row consists from left to right of the following sequence of blocks whose origins are located at the following coordinates:

      \[
      \begin{array}{cccc}
        x & \vdots & \vdots & \vdots \\
        \rightarrow x & x + (1 + 1 + t_i)W & x + (1 + t_{i+1})W & x + t_{i+2}W \\
        \vdots & \vdots & \vdots & \vdots \\
        \rightarrow \text{Marker} & i + 2 + t_{i+1} & i + 2 + t_{i+1} & i + 2 + t_{i+1} \\
        \end{array}
      \]

    This row ends at position $((i + 1)h + (1 + |s| + |\alpha^{1+i+t_{i+1}}| + t_{i+1})W - 7, 2ih + 2)$.  
    - the zag-row consists from right to left of the following sequence of blocks whose origins are located at the following coordinates:

      \[
      \begin{array}{cccc}
        y & \vdots & \vdots & \vdots \\
        \rightarrow y & x + (1 + 1 + t_i)W & x + (1 + t_{i+1})W & x + t_{i+2}W \\
        \vdots & \vdots & \vdots & \vdots \\
        \rightarrow \text{Marker} & i + 2 + t_{i+1} & i + 2 + t_{i+1} & i + 2 + t_{i+1} \\
        \end{array}
      \]

    where $v = u^{1+t_{i+1}} = s \cdot \alpha^{1+i+t_{i+1}} \neq \epsilon$ (as $s$ and $\alpha^{1+i+t_{i+1}}$ are not both $\epsilon$). This row ends at position $((i + 1)h + (1 + t_{i+1})W - 1, 2(i + 1)h)$.
  - (Halt 1) if $u^{1+t_i} = 0^*1$ and $\alpha^{1+i+t_{i+1}} = \epsilon$: then $r = t_{i+1} - t_i - 1$ and the last rows of the configuration consists from left to right of the following sequence of blocks located at the following coordinates:

      \[
      \begin{array}{cccc}
        x & \vdots & \vdots & \vdots \\
        \rightarrow x & x + (1 + t_i)W & x + (t_{i+1} - 1)W & x + t_{i+2}W \\
        \vdots & \vdots & \vdots & \vdots \\
        \rightarrow \text{Marker} & i + 1 + t_i & i + t_{i+1} & i + 2 + t_{i+1} \\
        \end{array}
      \]

    where $v = u^{1+t_{i+1}} = \epsilon$ and $\alpha^{1+i+t_{i+1}} = \epsilon$. \text{Halt}.
  - (Halt 2) if $u^{1+t_i} = 0^*$ for some $r \geq 0$: then the $i$-th zig-row is last row of the configuration and consists from left to right of the following sequence of blocks located at the following coordinates:

      \[
      \begin{array}{cccc}
        x & \vdots & \vdots & \vdots \\
        \rightarrow x & x + (1 + t_i)W & x + (t_i + r)W & x + (1 + r + t_i)W \\
        \vdots & \vdots & \vdots & \vdots \\
        \rightarrow \text{Marker} & i + 1 + t_i & i + r + t_i & i + r + 1 + t_i \\
        \end{array}
      \]


The following sections are dedicated to the proof of Key Lemma 3.

4 Advanced Design Tool box

In this section, we present several key tools to program Oritatami design and which we believe to be generic as they allowed us to get a lot of freedom in our design.
4.1 Implementing the logic

As in [6], the internal state of our “molecular computing machinery” consists essentially of two parameters: 1) the position inside the transcript of the part currently folding; and 2) the entry point of transcript inside the environment. Indeed, depending on the entry point or the position inside the transcript, different beads will be in contact with the environment and thus different functions will be applied as a result of their interactions. This happens during the zig phase: in the first (zig-up) part, the transcript starts folding at the bottom, forcing the modules $G\text{ Read }$ to fold into $G\text{ Copy }$ bricks; whereas during the second (zig-down) part, the transcript starts folding at the top, forcing the modules $G\text{ Copy }$ to fold into $G\text{ Read }$ bricks instead. Similarly, the memory of the system consists of the beads already placed on the surrounding of the area currently visited (the environment). This happens in every row of the folding: depending on the letter encoded at the bottom of the row above, the modules $G\text{ Copy }$ fold into $G\text{ Read }$ bricks (zig-up phase), $G\text{ Copy }$ bricks (zig-down phase), and $G\text{ Copy }$ bricks (zag phase).

At different places, we need the transcript to read information from the environment and trigger the appropriate folding. This is obtained through different mechanisms.

**Default folding.** By default, during the zig-up phase, $B$ is attracted to the left by $F$ and folds to the right only in presence of $E$ above. This allows to continue the folding only if the tape word is not empty or to halt it otherwise (see Figure 27 in [7]).

**Geometry obstruction.** A typical example is illustrated by $G$. During the zig-up phase where the absence of environment below the block $G\text{ Read }$ allows $G$ to fold downward at the beginning (see Figure 41 in [7]) which shift the transcript by 7 beads along $F$ resulting in $G$ to adopt the glider-shape (more details on this mechanism in the next section). Whereas during the zig-down phase, $G$ cannot make this loop because it is occupied by a previously placed $G$. This results in a perfect alignment of $G$ with $F$ whose strong attraction forces $G$ to adopt the switchback shape (see Figure 43 in [7]).

**Geometry of the environment.** Figure 3 shows how the shape of the environment is used to change the direction of $G$ in glider-shape. This results in modifying the entry point in the environment and allows the Oritatami system to trim the leading Os in the tape word by going back to the same entry point (Fig. 3(a)), switch from zip-up to zig-down phase when reading a 1 by opening the next block from the top (Fig. 3(b)), and from zag to zig-up phase when it has rewound to the beginning of the tape word, by getting down to the bottom of the next zig row (Fig. 3(c)).

4.2 Easing the design: getting the freedom you need

Several key tools allowed to ease considerably our design, and even in some cases to make it feasible. These tools are generic enough to be considered as programming paradigms.

One main difficulty we faced is that the different functions one wants to implement tend to concentrate at the same “hot-spots” in the transcript. A typical example is the midpoint of $G$ where one wants to implement all the functions: Read, Copy and Line Feed. The following powerful tools allow to overcome these difficulties:

**Socks** work by letting a glider/switchback module fold into a switchback turn conformation for some time when it would otherwise fold into a glider. Examples are given in Figure 4. They are easy to implement: indeed, the socks naturally adopt the same shape as the corresponding switchback turn and require thus no extra interfering bonds. They allow a lot of freedom in the design, for several reasons:
First, they simplify the design of important switchback part by lifting the need for implementing the glider configuration for that part, as shown in Figure 4(a).

Second, a glider naturally progresses at speed 1/3. Adding a sock allows us to slow its progression down to speed 1/5 for some time (see Fig. 4(b)) and therefore to realign them. We used that feature repeatedly to “shift” some modules: starting the folding at an initial speed-1 (i.e. straight line) and then compensating for that speed later on by introducing socks (see Fig. 4(b)). This is a key point in our design, as it allowed us to spread apart the Read and Copy functions into different subsequences of module $G$, and therefore to get less constraints on our rule design. In the specific case of module $G$, the Copy-function occurs at the center of the module, while the Read-function is implemented earlier in module! (see section F.10 in [7] for full details)

Finally, socks allow to prevent unwanted interactions between beads by concealing potentially harmful beads in an unreachable area as in Figure 4(c).

**Exponential bead type coloring** is a key tool to allow module $G$ to fold into different shapes, glider or switchback, along module $F$, when folding in the Read configuration. The problem it solves is that in order for $G$ to fold into switchbacks, we need strong interactions between $G$ and its neighboring module $F$ (see Fig. 41 in [7]), whereas in order for $G$ to fold as glider, we want to avoid those interactions (see Fig. 43 in [7]).
Figure 5: Two examples of proof trees for the same subsequence in two different environments. The number at the upper-left corner of every ball stands for the number of bonds for the path inside the ball. The number at the lower right corner of each ball stands for the number of paths grouped in the ball, allowing to check that no path was omitted. Balls highlighted in black bold contain the bonds-maximizing paths. Balls highlighted in grey bold contain the paths that place the bead at the same location as the bonds-maximizing paths, and which must thus be considered in the next level as well.

This is made possible because gliders progress at speed 1/3 while switchbacks progress at speed 1. Using a power-of-3 coloring, we manage to easily achieve these contradicting goals altogether (the construction is analysed in Lemma 11 in Section G.1 in [7]).

5 Correctness of local folding: Proof tree certificates

The goal of this section is to conclude the proof of our design by proving Key Lemma 3. The proof works by induction, assuming that the preceding beads of the transcript fold at the locations claimed by the lemma. We proceed in 3 steps:

- We first enumerate all the possible environments for every part of the transcript. As, we carefully aligned our design, most of the beads only see a small number of different environments.

- For the few cases (three in total) where the number of environments is unbounded, we give an explicit proof of correctness of their folding (Lemmas 9–11 in section G.1 in [7]). This is where the concealing feature of socks and the exponential bead type coloring play a crucial role.

- For all the other cases, we designed human-checkable computer-generated certificates, called proof trees. It consists in listing in a compact but readable manner all the possible paths for the transcript in every possible environment. In order to match human readability, paths with identical bonding patterns are grouped into one single ball. Balls containing the paths maximizing the number of bonds are highlighted in bold and organized in a tree. This reduces the number of cases to less than 5 balls in most of the levels of the tree, achieving human-checkability of the computed certificate (see Fig. 5). Proof trees are available at https://www.irif.fr/~nschaban/oritatami/.
References


Cluster Editing in Multi-Layer and Temporal Graphs

Jiehua Chen
Faculty of Mathematics, Informatics, and Mechanics, University of Warsaw, Poland
jiehua.chen2@gmail.com

Hendrik Molter
Algorithmics and Computational Complexity, Faculty IV, TU Berlin, Berlin, Germany
h.molter@tu-berlin.de

Manuel Sorge
Faculty of Mathematics, Informatics, and Mechanics, University of Warsaw, Poland
manuel.sorge@mimuw.edu.pl

Ondřej Suchý
Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, Prague, Czech Republic
ondrej.suchy@fit.cvut.cz

Abstract
Motivated by the recent rapid growth of research for algorithms to cluster multi-layer and temporal graphs, we study extensions of the classical Cluster Editing problem. In Multi-Layer Cluster Editing we receive a set of graphs on the same vertex set, called layers and aim to transform all layers into cluster graphs (disjoint unions of cliques) that differ only slightly. More specifically, we want to mark at most \(d\) vertices and to transform each layer into a cluster graph using at most \(k\) edge additions or deletions per layer so that, if we remove the marked vertices, we obtain the same cluster graph in all layers. In Temporal Cluster Editing we receive a sequence of layers and we want to transform each layer into a cluster graph so that consecutive layers differ only slightly. That is, we want to transform each layer into a cluster graph with at most \(k\) edge additions or deletions and to mark a distinct set of \(d\) vertices in each layer so that each two consecutive layers are the same after removing the vertices marked in the first of the two layers. We study the combinatorial structure of the two problems via their parameterized complexity with respect to the parameters \(d\) and \(k\), among others. Despite the similar definition, the two problems behave quite differently: In particular, Multi-Layer Cluster Editing is fixed-parameter tractable with running time \(k^{O(k+d)}s^{O(1)}\) for inputs of size \(s\), whereas Temporal Cluster Editing is \(W[1]\)-hard with respect to \(k\) even if \(d = 3\).

2012 ACM Subject Classification Theory of computation → Fixed parameter tractability

Keywords and phrases Cluster Editing, Temporal Graphs, Multi-Layer Graphs, Fixed-Parameter Algorithms, Polynomial Kernels, Parameterized Complexity

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.24

Related Version Preprint of the full version of this paper is available from ArXiv [7], https://arxiv.org/abs/1709.09100.
Cluster Editing in Multi-Layer and Temporal Graphs

Funding JC and MS supported by the People Programme (Marie Curie Actions) of the European Union’s Seventh Framework Programme (FP7/2007-2013) under REA grant agreement number 631163.11, the Israel Science Foundation (grant no. 551145/14), and by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme under grant agreement numbers 677651 (JC) and 714704 (MS). HM supported by the DFG, project MATE (NI 369/17). OS supported by grant 17-20065S of the Czech Science Foundation.

Acknowledgements Work initiated at the annual research retreat of the TU Berlin Algorithmics and Computational Complexity group held in Boiensdorf (Baltic Sea), in April 2017. Main work of JC and MS done while with Ben-Gurion University of the Negev, Beer Sheva, Israel.

1 Introduction

Cluster Editing and its weighted form Correlation Clustering are two important and well-studied models of graph clustering [1, 4, 6, 12, 18]. In the former, we are given a graph and we aim to edit (that is, add or delete) the fewest number of edges in order to obtain a cluster graph, a graph in which each connected component is a clique. Cluster Editing has attracted a lot of attention from a parameterized-algorithms point of view (e.g. [4, 6, 12, 18]) and the resulting contributions have found their way back into practice [4].

Meanwhile, additional information is now available and used in clustering methods. In particular, research on clustering so-called multi-layer and temporal graphs grows rapidly (e.g. [16, 22, 23, 24, 25]). A multi-layer graph is a set of graphs, called layers, on the same vertex set [5, 16, 17]. In social networks, a layer can represent social interactions, geographic closeness, common interests or activities [16]. A temporal graph is a multi-layer graph in which the layers are ordered linearly [14, 15, 19, 20, 24, 25]. Temporal graphs naturally model the evolution of relationships of individuals over time or their set of time-stamped interactions.

The goals in clustering multi-layer and temporal graphs are, respectively, to find a clustering that is consistent with all layers [16, 17, 22, 23] or a clustering that slowly evolves over time consistently with the graph [24, 25]. The methods used herein are often heuristic and beyond observing NP-hardness, to the best of our knowledge, there is no deeper analysis of the complexity of the general underlying computational problems that are attacked in this way. Hence, there is also a lack of knowledge about the possible avenues for algorithmic tractability. We initiate this research here.

We analyze the combinatorial structure behind cluster editing for multi-layer and temporal graphs, defined formally below, via studying their parameterized complexity with respect to the most basic parameters, such as the number of edits. That is, we aim to find fixed-parameter algorithms, which have running time $f(p) \cdot \ell^{O(1)}$ where $p$ is the parameter and $\ell$ the input length, or to show W[1]-hardness, which indicates that there cannot be such algorithms.

As we will see, both problems offer rich interactions between the layers on top of the structure inherited from Cluster Editing. Our main contributions are an intricate fixed-parameter algorithm for multi-layer cluster editing, whose underlying techniques should be applicable to a broader range of multi-layer problems, and a hardness result for temporal cluster editing, which shows that certain non-local structures harbor algorithmic intractability.

---

1 When considering the activity in different communities, we typically obtain a large number of layers [21].
Temporal Cluster Editing (TCE). Berger-Wolf and Tantipathananandh [25] were motivated by cluster detection problems from practice to study the following problem. Given a temporal graph, edit each layer into a cluster graph, that is, add or remove edges such that the layer becomes a disjoint union of cliques, while minimizing the total number of edits and the number of vertices moving between different clusters in two consecutive layers. TCE is a variant of this problem where instead minimize the layer-wise maxima of the number of edits and moving vertices, respectively. The problem can be formalized as follows.

Let \( G = (G_i)_{i \in [\ell]} \) be a temporal graph with vertex set \( V \), that is, \( G_i \) is the \( i \)-th layer. Let \( G_i = (V, E_i) \). An edge modification set for a graph \( G = (V, E) \) is a set of pairs of vertices from \( V \). A clustering for \( G \) is a sequence \( M = (M_i)_{i \in [\ell]} \) of edge modification sets such that each layer \( G_i \) is turned into a cluster graph \( G_i' = (V, E_i \oplus M_i) \).\(^2\) (Throughout this work, \( G_i' \) denotes the modified \( i \)-th layer of the temporal or multi-layer graph and the corresponding clustering understood from the context.) Intuitively, sets \( M_i \) contain the data that we need to disregard in order to cluster our input and hence we want to minimize their sizes [24, 25].

For that, we say that \( M \) is \( k \)-bounded for some integer \( k \in \mathbb{N} \) if \(|M_i| \leq k\) for each \( i \in \ell \).

A fundamental property of clusterings of temporal graphs is their evolution over time. In practice, these clusterings evolve only slowly as measured by the number of vertices switching between clusters from one layer to another [24, 25]. This requirement can be formalized as follows. Let \( d \in \mathbb{N} \). Clustering \( M \) for \( G \) (as above) is temporally \( d \)-consistent if there exists a sequence \((D_i)_{i \in [\ell - 1]} \) of vertex sets such that \( G_i'[V \setminus D_i] = G_i'[V \setminus D_i'] \) for each \( i \in [\ell - 1] \). Hence, the sets \( D_i \) contain the vertices changing clusters. We arrive at the following.

**Temporal Cluster Editing (TCE)**

**Input:** A temporal graph \( G \) and two integers \( k,d \).

**Question:** Is there a temporally \( d \)-consistent \( k \)-bounded clustering for \( G \)?

We also say that the corresponding sets \( D_i \subseteq V \) and \( M_i \subseteq \binom{V}{2} \) as above form a solution and the vertices in \( D_i \) are marked.

The most natural parameters are the “number \( k \) of edge modifications per layer”, the “number \( d \) of marked vertices”, the “number \( \ell \) of layers”, and the “number \( n = |V| \) of vertices”. An overview on our results is shown in Figure 1. (Note that, within these parameters, we have \( d \leq n \) and \( k \leq n^2 \).) A straightforward reduction yields that TCE is \( \mathsf{NP} \)-complete even if both \( d = 0 \) and \( \ell = 1 \) (\(^3\)). On the positive side, we obtain an algorithm for TCE with running time \( n^{O(k)}\ell \). The basic idea is to check whether any two possible cluster editing sets for two consecutive layers allow for a small number of marked vertices by matching techniques. As it turns out, even for \( d = 3 \), we cannot obtain an improved running time on the order of \((n \ell)^{o(k)}\) unless the Exponential Time Hypothesis (ETH) fails. The reason is an obstruction represented by small clusters which may have to be joined or split throughout many layers, to be able to form clusters in some later layer. Finally, we give a polynomial kernel with respect to the parameter combination \( (d, k, \ell) \) and show that the problem does not admit a polynomial kernel for parameter “number \( n \) of vertices” unless \( \mathsf{NP} \subseteq \mathsf{coNP}/\mathsf{poly} \).

Multi-Layer Cluster Editing (MLCE). For clusterings of multi-layer graphs we typically have to consider the tradeoff between closely matching individual layers and getting an

\(^2\) Herein, \( \oplus \) denotes the symmetric difference: \( A \oplus B = (A \setminus B) \cup (B \setminus A) \) and \( [\ell] \) denotes the set \( \{1, \ldots, \ell\} \) for \( \ell \in \mathbb{N} \).

\(^3\) The proofs of results marked by (\(*\)) and proofs of correctness and safeness of reduction rules and branching rules marked by (\(*\)) are omitted due to space constraints and deferred to a full version [7].
Cluster Editing in Multi-Layer and Temporal Graphs

<table>
<thead>
<tr>
<th>Parameter Combinations</th>
<th>Complexity</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>((n, \ell))</td>
<td>Instance size</td>
<td></td>
</tr>
<tr>
<td>((d, k, \ell))</td>
<td>FPT, No poly kernel [Thm 11 &amp; Prop 15]</td>
<td>(k) \text{-hard even for } (d = 3) [Thm 12]</td>
</tr>
<tr>
<td>((d, k))</td>
<td>Poly kernel [Thm 13]</td>
<td>(W[1])-hard</td>
</tr>
<tr>
<td>((k, \ell))</td>
<td>open</td>
<td>para-NP-hard (⋆)</td>
</tr>
<tr>
<td>((d, \ell))</td>
<td>para-NP-hard (⋆)</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 1** Our results for TCE and MLCE in a Hasse diagram of the upper-boundedness relation between the parameters the “number \(k\) of edge modifications per layer”, the “number \(d\) of marked vertices”, the “number \(\ell\) of layers”, and the “number \(n = |V|\) of vertices” and all of their combinations. A node is split into two parts if the complexity results differ; the left part shows the result for TCE, the right part for MLCE. Red entries mean that the corresponding parameterized problem is para-NP-hard. Orange entries mean that the corresponding parameterized problem is \(W[1]\)-hard while contained in XP. It is in FPT for all parameter combinations colored yellow or green and admits a polynomial kernel for all parameter combinations colored green. It does not admit a polynomial kernel for all parameter combinations that are colored yellow unless \(\text{NP} \subseteq \text{coNP}/\text{poly}\). A tight parameterized complexity classification for the gray colored parameter combination is open.

To briefly summarize our results for MLCE: While strong overall fit (small parameter \(d\)) or closely matched layers (small parameter \(k\)) alone do not lead to fixed-parameter tractability, jointly they do. Indeed, we obtain an \(O(k + d) \cdot n^3 \cdot \ell\)-time algorithm, in contrast to TCE. At first glance, this is surprising because in the temporal case, we only need to satisfy the consistency condition “locally”. This requires less interaction among layers and thus, seemed to be easier to tackle than the multi-layer case. The algorithm uses a novel method that allows us make decisions over a large number of layers at once. It can be compared with greedy localization [9] in that some of the decisions are greedy and transient, meaning that they seem intuitively favorable and can be reversed in individual layers if they later turn out to be wrong. However, the application of this method is not straightforward, requires new techniques to deal with the interaction between layers and consequently intricately tuned branching and reduction rules.

---

\(^4\) Below we drop the qualifiers “temporally” and “totally” if they are clear from the context.
Algorithm 1: MLCE.

Input:
- A set of graphs $G_1, \ldots, G_\ell = (V, E_1), \ldots, (V, E_\ell)$ two integers $k$ and $d$.
- A set of marked vertices $D$, edge modification sets $M_1, \ldots, M_\ell$.
- A set $B \subseteq (V \setminus D)$ of permanent vertex pairs.

1. if $|D| > d$ or there is an $i \in [\ell]$ such that $|M_i \cap B| > k$ then return false
2. Apply the first applicable rule in the following ordered list: 1, Greedy Rule, 2, Clean-up Rule, 3, and 4.
3. return true

We in fact completely classify MLCE in terms of fixed-parameter tractability and existence of polynomial-size problem kernels with respect to the parameters $k, d, \ell$, and $n$, and all of their combinations, see Figure 1 for an overview. MLCE is para-NP-hard for all parameter combinations which are smaller or incomparable to $k + d$. Straightforward reductions yield NP-completeness even if both $d = 0$ and $\ell = 1$ or both $k = 0$ and $\ell = 3$; the problem is polynomial-time solvable if $k = 0$ and $\ell \leq 2 \,(\ast)$. Finally, the kernelization results for TCE also hold for MLCE, that is, the problem admits a polynomial kernel with respect to $(d, k, \ell)$ and does not admit a polynomial kernel for the “number $n$ of vertices” unless NP $\subseteq$ coNP/poly.

Related Work. We are not aware of studies of the fundamental algorithmic properties of multilayer and temporal graph clustering. In terms of parameterized algorithms, only the indirect approach of aggregating clusterings into one has been studied for multilayer [3, 11] and temporal graphs [24]. These approaches are less accurate, however [2, 25]. The approximability of temporal versions of $k$-means clustering and its variants was studied by Dey et al. [10].

2 Multi-Layer Cluster Editing (MLCE)

In this section, we show that MLCE can be solved efficiently for small $k$ and $d$.

Theorem 1. MLCE is FPT with respect to the number $k$ of edge modifications per layer and number $d$ of marked vertices combined. It can be solved in $k^{O(k+d)} \cdot n^3 \cdot \ell$ time.

We describe a recursive search-tree algorithm (see algorithm 1) for the following input:

- An instance $I$ of MLCE consisting of a multi-layer graph $G_1, \ldots, G_\ell = (V, E_1), \ldots, (V, E_\ell)$ and two integers $k$ and $d$.
- A constraint $P = (D, (M_i)_{i \in [\ell]}, B)$, consisting of a set of marked vertices $D \subseteq V$, edge modification sets $M_1, \ldots, M_\ell \subseteq (V \setminus D)$, and a set $B \subseteq (V \setminus D)$ of permanent vertex pairs.

The algorithm follows the greedy localization approach [9] in which we make some decisions greedily, which we possibly revert through branching later on. The greedy decisions herein give us some structure that we can exploit to keep the search-tree size small. The edge modification sets $M_i$ represent both the greedy decisions and those that we made through branching. The set $B$ contains only those made by branching.

Throughout the algorithm, we try to maintain a property that the constraint at hand is good which intuitively means that the constraint can be turned into a solution (if one exists).
Definition 2 (Good Constraint). Let $I$ be an instance of MLCE. A constraint $P = (D, M_1, \ldots, M_\ell, B)$ is good for $I$ if there is a solution $S = (M^*_1, \ldots, M^*_\ell, D^*)$ such that (i) $D \subseteq D^*$, (ii) there is no $\{u, v\} \in B$ such that $u \in D^*$, and (iii) for all $i \in [\ell]$ we have $M_i \cap B = M^*_i \cap B$. We also say that $S$ witnesses that $P$ is good.

Furthermore, it is easy to see that an “empty” constraint is good.

Observation 3. For any yes-instance $I = (G_1, \ldots, G_\ell, d, k)$ of MLCE, we have that $P_0 = (D = \emptyset, M_1 = \emptyset, \ldots, M_\ell = \emptyset, B = \emptyset)$ is a good constraint.

We also call the above constraint $P_0$ trivial. The initial call of our algorithm is with the input instance of MLCE together with the trivial constraint $P_0$.

Our algorithm uses various different branching rules to search for a solution to an MLCE input instance: A branching rule takes as input an instance $I$ of MLCE and a constraint $P$ and returns a set of constraints $P^{(1)}, \ldots, P^{(x)}$. When a branching rule is applied, the algorithm invokes a recursive call for each constraint returned by the branching rule and returns true if at least one of the recursive calls returns true; otherwise, it returns false. For that to be correct, whenever a branching rule is invoked with a good constraint, at least one of the constraints returned by the branching rule has to be a good constraint as well. In this case we say that a branching rule is safe.

In the following, we introduce the branching rules used by the algorithm and prove that each of them is safe. This together with Theorem 3 will allow us to prove by induction that the algorithm eventually finds a solution for the input instance of MLCE if it is a yes-instance. To make the description of the branching rules more readable, we introduce four types of non-marked vertex pairs. Say that a vertex pair $\{u, v\} \in \binom{V}{2}$

- is settled if $\{u, v\} \in E_i \cup M_i$ for all $i \in \ell$ or $\{u, v\} \notin E_i \cup M_i$ for all $i \in [\ell]$ (edge always present or never present),
- is frequent if $|\{|i | \{u, v\} \in E_i \cup M_i\}| \geq \frac{2\ell}{3}$ (edge almost always present),
- is scarce if $|\{|i | \{u, v\} \in E_i \cup M_i\}| \leq \frac{\ell}{3}$ (edge almost never present), and
- is unsettled otherwise, that is, $\frac{\ell}{3} < |\{|i | \{u, v\} \in E_i \cup M_i\}| < \frac{2\ell}{3}$ (edge sometimes present).

Note that, if a vertex pair $\{u, v\}$ falls in one of the above categories, both $u, v$ are not marked.

Our aim with the first two rules is to settle all pairs in $\binom{V}{2}$. In order to achieve our running time bound, we can only afford to exhaustively search through all unsettled vertex pairs:

Branching Rule 1 (*). If there is an unsettled vertex pair $\{u, v\} \in \binom{V}{2}$, then output the following up to four constraints:

1. For all $i \in [\ell]$, put $M^{(1)}_i = M_i \cup \{\{u, v\} \setminus E_i\}$, $D^{(1)} = D$, and $B^{(1)} = B \cup \{\{u, v\}\}$.
2. For all $i \in [\ell]$, put $M^{(2)}_i = M_i \cup \{\{u, v\} \cap E_i\}$, $D^{(2)} = D$, and $B^{(2)} = B \cup \{\{u, v\}\}$.
3. If there is no $x \in V \setminus D$ with $\{u, x\} \in B$, then $D^{(3)} = D \cup \{u\}$, the rest stays the same.
4. If there is no $x \in V \setminus D$ with $\{v, x\} \in B$, then $D^{(4)} = D \cup \{v\}$, the rest stays the same.

The following Greedy Rule deals with all frequent and scarce vertex pairs. It only produces one constraint and hence no branching occurs in that sense. For formal reasons it is nevertheless useful to treat the Greedy Rule as a special case of a branching rule. Note that the algorithm also invokes a recursive call with the output constraint of this rule. The rule greedily adds the edge corresponding to a frequent vertex pair in all layers where it is not present and removes edges corresponding to scarce vertex pairs in all layers where it is present. Intuitively, the Greedy Rule is safe, because all of its decisions can be reverted later.

Greedy Rule (*). If there is a frequent or a scarce vertex pair $\{u, v\} \in \binom{V}{2}$, then return one of the following two constraints:
With the following rule we edit the subgraphs induced by all non-marked vertices into
If none of the above possibilities apply, then reject the current branch.
This technically does not fit the definition of a branching rule but we can achieve the same effect by
following constraints:
Branching Rule 3 requires that 1 and the Greedy Rule are not applicable.
For technical reasons, it also
these choices. Also, to have a correct estimate of the sizes of the current edge modification
search space, we expect that some of the choices were not correct. This rule will then revert
Greedy Rule we greedily make decisions and do not exhaustively search through the whole
return a constraint with
one constraint and hence no branching occurs, so it is also a degenerate branching rule. Note
that the algorithm also invokes a recursive call with the output constraint of this rule.
Clean-up Rule (⋆). If there is an i ∈ [ℓ] such that there is a {u, v} ∈ Mi with u ∈ D, then return a constraint with M′(1) = Mi \ {{u, v}}, the rest stays the same.

The next rule tries to repair any budget violations that might occur. Since with the
Greedy Rule we greedily make decisions and do not exhaustively search through the whole
search space, we expect that some of the choices were not correct. This rule will then revert
these choices. Also, to have a correct estimate of the sizes of the current edge modification
sets, this rule requires that the Clean-up Rule is not applicable. For technical reasons, it also
requires that 1 and the Greedy Rule are not applicable.
Branching Rule 3 (⋆). If there is an Mi for some i ∈ [ℓ] with |Mi| > k, then if |Mi \ B| ≤ k + 1, let M′i = Mi \ B, otherwise, take any M′i ⊆ Mi \ B with |M′i| = k + 1 and return the following constraints:
1. For each {u, v} ∈ M′i return a constraint in which for all j ∈ [ℓ] we put M′(1)j = Mj ∪ {{u, v}}, D′(1) = D, and B′(1) = B ∪ {{u, v}}.
2. For each {u, v} ∈ M′i:
   - If there is no x ∈ V \ D such that {u, x} ∈ B, then return a constraint with D′(1) = D ∪ {u}, B′(1) = B, and 1 ≤ j ≤ ℓ: M′(1)j = Mj \ {{u, v}}.
   - If there is no x ∈ V \ D such that {v, x} ∈ B, then return a constraint with D′(1) = D ∪ {v}, B′(1) = B and 1 ≤ j ≤ ℓ: M′(1)j = Mj \ {{u, v}}.
If M′i = ∅, then reject the current branch.

5 This technically does not fit the definition of a branching rule but we can achieve the same effect by returning trivially unsatisfiable constraints such as a constraint with |D′(1)| > d.
The last rule, 4, requires that all other rules are not applicable. In this case the non-marked vertices induce the same cluster graph in every layer. 4 checks whether in every layer it is possible to turn the whole layer (including the marked vertices) into a cluster graph such that the cluster graph induced by the non-marked vertices stays the same and the edge modification budget is not violated in any layer. If this is not the case for a layer \( i \), the rule checks whether it is necessary to revert a greedy decision or whether there is an induced \( P_3 \) where exactly one vertex is marked and it is necessary to modify the vertex pair not containing the marked vertex. To achieve the latter we introduce a modified version of a known kernelization algorithm \([13]\) for classic Cluster Editing. We call the algorithm \( K \) and it takes as input a tuple \((G_i, k, D, M_i, B)\) and either outputs a distinct failure symbol or two sets \( R \) and \( C \), where \( R \) contains all unmarked vertex pairs modified by \( K \) and \( C \) contains unmarked vertex pairs of the produced kernel that are part of induced \( P_3 \)s. The formal description is as follows.

**Modified Kernelization Algorithm \( K \).** Given an input \((G_i, k, D, M_i, B)\). First, set all vertex pairs in \( M_i \cup B \) to obligatory and exhaustively apply the following modified versions of standard data reduction rules for Cluster Editing to \( G_i' = (V, E_i \oplus M_i) \). Let \( k_i = k - |M_i| \) and \( R = \emptyset \).

- If \( k_i < 0 \) or there is an induced \( P_3 \) where all vertex pairs are obligatory, then abort and output a failure symbol.
- If a vertex pair \( \{u, v\} \) is involved in \( k_i + 1 \) induced \( P_3 \)'s, then, if it is obligatory, abort and output a failure symbol, otherwise modify it, set it to obligatory, and reduce \( k_i \) by one. If \( u \notin D \) and \( v \notin D \), then add \( \{u, v\} \) to \( R \).
- If there is an isolated clique, then remove it.

Let \( G_i'(R) \) be the reduced version of \( G_i' \). If the number of vertices in \( G_i'(R) \) is larger than \( k_i^2 + 2k_i \), then abort and output a failure symbol. Otherwise, let \( C \) be the set of all vertex pairs of unmarked (not contained in \( D \)) vertices that are part of an induced \( P_3 \) in \( G_i'(R) \). Output \( R \) and \( C \).

**Branching Rule 4 (⋆).** For all \( 1 \leq i \leq \ell \) we use \( M_i \) to denote the set of all possible edge modifications where each edge is incident to at least one marked vertex, that turn \( G_i' = (V, E_i \oplus M_i) \) into a cluster graph. More specifically, we have

\[
M_i = \{ M \subseteq \binom{V}{2} \mid \forall e \in M : e \cap D \neq \emptyset \land G_i'' = (V, E_i \oplus (M_i \cup M)) \text{ is a cluster graph} \}.
\]

If there is an \( 1 \leq i \leq \ell \) such that \( \min_{M \in M_i} |M| > k - |M_i| \) then let \( M_i' = M_i \setminus B \) and invoke the modified kernelization algorithm \( K \) on \((G_i, k, D, M_i, B)\). If it outputs a failure symbol and \( M_i' = \emptyset \), then reject the current branch. Otherwise let \( R \) and \( C \) be the sets output by \( K \) or \( R = C = \emptyset \) if \( K \) output a failure symbol, and return the following constraints:

1. For each \( \{u, v\} \in M_i' \):
   - Return a constraint in which for all \( j \in [\ell] \) we put \( M_j^{(1)} = M_j \oplus \{u, v\} \), \( D^{(1)} = D \), and \( B^{(1)} = B \cup \{u, v\} \).
   - If there is no \( x \in V \setminus D \) such that \( \{u, x\} \in B \), then return a constraint with \( D^{(1)} = D \cup \{u\} \), \( B^{(1)} = B \), and \( 1 \leq j \leq \ell : M_j^{(1)} = M_j \setminus \{u, v\} \).
   - If there is no \( x \in V \setminus D \) such that \( \{v, x\} \in B \), then return a constraint with \( D^{(1)} = D \cup \{v\} \), \( B^{(1)} = B \), and \( 1 \leq j \leq \ell : M_j^{(1)} = M_j \setminus \{u, v\} \).
2. For each \( u \in V \setminus D \) such that \( \{u, v\} \in R \) for some \( v \in V \):
   - If there is no \( x \in V \setminus D \) such that \( \{u, x\} \in B \), then return a constraint with \( D^{(1)} = D \cup \{u\} \), \( B^{(1)} = B \), and \( 1 \leq j \leq \ell : M_j^{(1)} = M_j \).
   - If \( R \neq \emptyset \), then output a constraint with \( D^{(1)} = D \), \( B^{(1)} = B \cup M_i \cup R \), and \( 1 \leq j \leq \ell : M_j^{(1)} = M_j \oplus R \).
3. For each \( \{u, v\} \in C \):
   - If there is no \( x \in V \setminus D \) such that \( \{u, x\} \in B \), then return a constraint with \( D^{(1)} = D \cup \{u\} \), and the rest stays the same. If there is no \( x \in V \setminus D \) such that \( \{v, x\} \in B \), then return a constraint with \( D^{(1)} = D \cup \{v\} \), and the rest stays the same.
   - Return a constraint with \( D^{(1)} = D \), \( B^{(1)} = B \cup M_1 \cup R \cup \{\{u, v\}\} \), and \( 1 \leq j \leq \ell \):
     \[
     M_j^{(1)} = M_j \oplus R \oplus \{\{u, v\}\}.
     \]

We now prove the correctness of the algorithm. By a straightforward analysis of the branching rules it follows that, if none is applicable, then the current constraint \( P \) yields a solution. Hence, whenever the algorithm outputs \texttt{true}, then the input is indeed a yes-instance.

\begin{itemize}
  \item \textbf{Lemma 4 (⋆).} Given an instance \( I \) of MLCE, if algorithm 1 outputs \texttt{true} on input \( I \) and the trivial partial solution \( P_0 \), then \( I \) is a yes-instance.
\end{itemize}

To show that, whenever the input instance \( I \) of the algorithm is a yes-instance, then the algorithm outputs \texttt{true}, we define the \textit{quality} of a good constraint and show that the algorithm increases the quality until it eventually finds a solution.

\begin{itemize}
  \item \textbf{Definition 5 (Quality of a constraint).} Let \( I = (G_1, \ldots, G_{\ell}, k, d) \) be an instance of MLCE. The \textit{quality} \( \gamma_I(P) \) of a constraint \( P = (D, M_1, \ldots, M_{\ell}, B) \) for \( I \) is 
    \[
    \gamma_I(P) = |D| + |B| - |\{\{u, v\} \in (V^{(D)}_1) \mid \{u, v\} \text{ is frequent or scarce}\}|.
    \]
  \item \textbf{Lemma 6 (⋆).} Let \( P \) be a good constraint for a yes-instance of MLCE. If applicable, each of the Greedy Rule and Branching Rules 1, 2, 3, and 4 return a good constraint with strictly increased quality in comparison to \( P \).
\end{itemize}

Next we show that the notion of quality of a good constraint is indeed a measure that allows us to argue that the algorithm eventually produces a solution (if it exists).

\begin{itemize}
  \item \textbf{Lemma 7 (⋆).} Let \( I \) be a yes-instance of MLCE, then there is a constant \( c_I \geq 0 \) such that for every good constraint \( P \) we have that 
    \[
    \gamma_I(P) \leq c_I \quad \text{and there is at least one good constraint} \quad P_{\text{max}} \quad \text{with} \quad \gamma_I(P_{\text{max}}) = c_I.
    \]
    Furthermore, for any good constraint \( P'_{\text{max}} \) with \( \gamma_I(P'_{\text{max}}) = c_I \), we have that algorithm 1 outputs \texttt{true} on input \( I \) and \( P'_{\text{max}} \).
\end{itemize}

We can now show the correctness of algorithm 1. Theorem 4 ensures that we only output \texttt{true} if the input is actually a yes-instance and Lemmas 6 and 7 together with the safeness of all branching rules ensures that if the input is a yes-instance, the algorithm outputs \texttt{true}.

\begin{itemize}
  \item \textbf{Corollary 8 (Correctness of algorithm 1) (⋆).} Given a MLCE instance \( I \), algorithm 1 outputs \texttt{true} on input \( I \) and the trivial good constraint \( P_0 \) if and only if \( I \) is a yes-instance.
\end{itemize}

It remains to show that algorithm 1 has the claimed running time upper-bound. We can check that all branching rules create at most \( O(k^4) \) recursive calls. The differentiation between unsettled, frequent and scarce vertex pairs ensures that the edge modification sets in sufficiently many layers increase for the search tree to have depth of at most \( O(k + d) \). The time needed to apply a branching rule is dominated by 4, where we essentially have to solve classical \textsc{Cluster Editing} in every layer.

\begin{itemize}
  \item \textbf{Lemma 9 (⋆).} The running time of algorithm 1 is in \( k^{O(k+d)} \cdot O(n^3 \cdot \ell) \).
\end{itemize}
3 Temporal Cluster Editing (TCE)

In this section we provide an algorithm for TCE with a running time $n^{O(k)\ell}$ and show that the running time cannot substantially be improved unless the Exponential Time Hypothesis (ETH) fails. The algorithm uses the following algorithm for the two-layer case as a subroutine.

Theorem 11 implies that $\exists$ a polynomial kernel for $TCE$ can be solved in $O(n^2 \log n)$ time, where $n$ denotes the number of vertices.

Proof. Given an instance $(G_i)_{i \in [\ell]}$, $k, d)$ of TCE, build an $\ell$-partite graph $G$ as follows: For each possible cluster editing set of $G_i$ of size at most $k$, add a vertex to the $i$th part of $G$. Note that $G$ contains $O(n^{2k}\ell)$ vertices since each part contains $O(n^{2k})$ vertices. For each $i$, $1 \leq i \leq n - 1$, and each pair of vertices $u, v$ in $G$ such that $u$ is in part $i$ and $v$ is in part $(i + 1)$ add to $G$ the edge $(u, v)$ if the algorithm of Theorem 10 accepts on input of the following instance of MLCE. Let $M_u, M_v$ be the cluster editing sets corresponding to $u$ and $v$, respectively. The MLCE instance consists of a multi-layer graph with the two layers $(V, E_i \oplus M_u)$ and $(V, E_i \oplus M_v)$, edit budget equal to zero, and marking budget equal to $d$. For each pair of vertices $u, v$, constructing the corresponding MLCE instance and solving it takes $O(n^{2k}\ell)$ time, amounting to overall $O(\ell \cdot n^{4k+2}\log n)$ time, because there are at most $n^{4k}\ell$ pairs of vertices to consider. Finally, we test whether there is a path from a vertex in the first part to a vertex in the last part in $G$. As there are at most $n^{4k}\ell$ edges in $G$, this takes $O(n^{4k}\ell)$ time. Hence, overall the running time is $O(\ell \cdot n^{4k+3}\log n)$. The correctness is deferred to a full version [7].

Theorem 11 implies that TCE is fixed-parameter tractable when parameterized by the number $n$ of vertices. At first glance, it seems wasteful to iterate over all possible cluster editing sets for each layer. Rather, the interaction between two consecutive layers seems to be limited by $k$ and $d$, since the necessary edits are local to induced $P_3$, and the necessary markings are local to incongruent clusters (perhaps resulting from destroying $P_3$). However, to our surprise, when the number of layers grows, this interaction spirals out of control. As the reduction of the following hardness result implies, we have to take into account splitting up small clusters in an early layer (even though locally they were already cliques), so as to be able to form cluster graphs a large number of layers later on. This behavior stands in stark contrast to MLCE, where the combinatorial explosion is limited to $k$ and $d$.

Theorem 12 (∗). TCE is W[1]-hard with respect to $k$, even if $d = 3$. Moreover, it does not admit an $f(k)(n\ell)^{O(k)}$-time algorithm unless the ETH fails.

4 Kernelization for MLCE and TCE

In this section we investigate the kernelizability of MLCE and TCE for different combinations of the four parameters as introduced in section 1. More specifically, we identify the parameter combinations for which MLCE and TCE admit polynomial kernels, and then we identify the parameter combinations for which no polynomial kernels exist, unless $NP \subseteq coNP/poly$.

We first present a polynomial kernel for MLCE for the parameter combination $(k, d, \ell)$ and then argue that essentially the same reduction rules give a polynomial kernel for TCE.

Theorem 13. MLCE admits a kernel of size $O(\ell^3 \cdot (k + d)^4)$ and TCE admits a kernel of size $O(\ell^3 \cdot (k + d \cdot \ell)^4)$. Both kernels can be computed in $O(\ell \cdot n^3)$ time.
We provide several reduction rules that subsequently modify the instance and assume that if a particular rule is to be applied, then the instance is reduced with respect to all previous rules, that is, all previous rules were already exhaustively applied. We introduce **Multi-Layer Cluster Editing with Separate Budgets (MLCEwSB)** which differs from MLCE only in that, instead of a global upper bound \( k \) on the number of edits, we receive \( \ell \) individual budgets \( k_i \), \( i \in [\ell] \), and we require that \( |M_i| \leq k_i \).

We first transform the input instance of MLCE to an equivalent instance of MLCEwSB by letting \( k_i = k \) for every \( i \in [\ell] \). Then we apply all our reduction rules to MLCEwSB. Finally, we transform the resulting instance of MLCEwSB to an equivalent instance of MLCE with just a small increase of the vertex set. Through the presentation, let \((G_1 = (V, E_1)), \ldots, G_\ell = (V, E_\ell), k_1, \ldots, k_\ell, d) \) be the current instance and \( k = \max\{k_i \mid i \in [\ell]\} \).

Next, we apply slightly modified versions of well known rules for classical **Cluster Editing** [13] and apply them on each layer individually (⋆). These rules are known to produce a kernel of size \( k^2 + 2k \). Notably, we leave out a rule that removes isolated cliques. Hence, after the application of these rules we either conclude that we face a no-instance or every layer \( i \) consists of a set \( R_i \subseteq V \), that contains the vertices \( v \) that appear in some induced \( P_3 \) in \( G_i \), and a number of isolated cliques. Furthermore, let \( R = \bigcup_{i=1}^{\ell} R_i \).

As a major difference to **Cluster Editing** for a single layer, we cannot simply remove the vertices that are not involved in any \( P_3 \) since we require the cluster graphs in individual layers not to differ too much. Only vertices in the clusters that do not change can be removed.

**Reduction Rule 1 (⋆).** If there is a subset \( A \subseteq V \setminus R \) such that for each layer \( i \in [\ell] \), the subset \( A \) is the vertex set of a connected component of \( G_i \), then remove \( A \) (and the corresponding edges) from every \( G_i \).

The next rule allows us to reduce vertices that appear in exactly the same clusters.

**Reduction Rule 2 (⋆).** If there is a set \( A \subseteq V \setminus R \) with \( |A| \geq k + d + 3 \) such that for every layer \( i \in [\ell] \) it holds that all vertices of \( A \) are in the same connected component of \( G_i \), then select an arbitrary \( v \in A \) and remove \( v \) from every \( G_i \).

The next rule shows that the remaining clusters in a yes-instance cannot be too large.

**Reduction Rule 3 (⋆).** If there is a layer \( i \in [\ell] \) and a connected component \( A \) of \( G_i \) with \( |A \setminus R| \geq k + 2d + 3 \), then answer NO.

Now we introduce our final rule bounding the number of vertices in the instance.

**Reduction Rule 4 (⋆).** If \( |V| > \ell \cdot (k^2 + 2k + d \cdot (k + 2d + 2) + 2k) \), then answer NO.

After bounding the size of the instance through 4 it remains to transform the resulting instance of MLCEwSB to an equivalent instance of MLCE. To this end we introduce new vertex set \( A \) of size exactly \( 2k + 2 \) to \( V \) and to each \( E_i \) introduce all edges from \( \binom{A}{2} \). Then, for each \( i \in \{1, \ldots, \ell\} \) we remove \( k - k_i \) arbitrary edges between vertices of \( A \) from \( E_i \) and set \( k_i = k \). It is straightforward to show that this produces an equivalent instance, which can be turned into an equivalent instance of MLCE in an obvious way.

Since no rule increases \( k \), \( d \), or \( \ell \), \( |V| = O(\ell \cdot (k + d)^2) \), the resulting instance can be described using \( O(\ell^3 \cdot (k + d)^4) \) bits and it is equivalent to the original instance, it remains to show that the kernelization is computable in polynomial time.

**Lemma 14 (⋆).** The kernelization can be done in \( O(\ell \cdot n^3) \) time.
Lastly, we argue that slightly modified reduction rules can be applied to TCE (with individual edge-modification budgets, where the resulting instance can be transformed back). Intuitively this follows from the following observations: The reduction rules do not mark vertices, and the union of all marked vertices of a TCE solution together with the edge modification sets forms a solution for a MLCE instance, where the maximal number of marked vertices is \( d \cdot \ell \). Hence, replacing \( d \) with \( d \cdot \ell \) in the description of all reductions rules yields a set of rules that produce a kernel of size \( O(\ell^3 \cdot (k + d \cdot \ell)^4) \) for TCE (⋆).

In contrast, we have the following.

**Proposition 15 (⋆).** MLCE and TCE do not admit polynomial kernels with respect to the number \( n \) of vertices, unless \( \mathsf{NP} \subseteq \mathsf{coNP}/\mathsf{poly} \).

## 5 Conclusion

Our results highlight that TCE and MLCE are much richer in structure than classical Cluster Editing. Techniques for the classical problem seem to only carry over somewhat for kernelization algorithms and otherwise new methods are necessary. In this regard, we contribute our fixed-parameter algorithm for MLCE with respect to the combination of \( k \) and \( d \). In contrast, the \( \mathsf{W}[1] \)-hardness for TCE with respect to \( k \) for \( d = 3 \) highlights the obstacles we need to overcome. Perhaps we can break the temporal non-locality by bounding the number of allowed modifications at one vertex in any interval of layers of some fixed size.

## References


Parameterized Query Complexity of Hitting Set Using Stability of Sunflowers

Arijit Bishnu  
Indian Statistical Institute, Kolkata, India

Arijit Ghosh  
The Institute of Mathematical Sciences, Chennai, India

Sudeshna Kolay  
Eindhoven University of Technology, Eindhoven, Netherlands

Gopinath Mishra  
Indian Statistical Institute, Kolkata, India

Saket Saurabh  
The Institute of Mathematical Sciences, Chennai, India

Abstract

In this paper, we study the query complexity of parameterized decision and optimization versions of Hitting-Set. We also investigate the query complexity of Packing. In doing so, we use generalizations to hypergraphs of an earlier query model, known as BIS introduced by Beame et al. in ITCS’18. The query models considered are the GPIS and GPISE oracles. The GPIS and GPISE oracles are used for the decision and optimization versions of the problems, respectively. We use color coding and queries to the oracles to generate subsamples from the hypergraph, that retain some structural properties of the original hypergraph. We use the stability of the sunflowers in a non-trivial way to do so.

2012 ACM Subject Classification  Theory of computation → Fixed parameter tractability, Theory of computation → Streaming, sublinear and near linear time algorithms

Keywords and phrases  Query complexity, Hitting set, Parameterized complexity

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.25

Related Version  A full version of the paper is available at [3], https://arxiv.org/abs/1807.06272.

Introduction

In query complexity models for graph problems, the aim is to design algorithms that have access to the vertices $V(G)$ of a graph $G$, but not the edge set $E(G)$. Instead, these algorithms construct local copies by using oracles to probe or infer about a property of a part of the graph. Due to the lack of knowledge about global structures, often it is difficult to design algorithms even for problems that are classically known to have polynomial time algorithms.

A natural optimization question in this model is to minimize the number of queries to the oracle to solve the problem. The most generic approach towards this is to ask as few queries to the oracle before the local copy of the graph is an equivalent sample of the actual graph. This spawns the study of query complexity. The query complexity of the algorithm is the number of queries made to the oracle. Keeping this in mind, several query models have been designed through the years. Let us take the example of the problem of finding a global minimum cut that has led to the introduction of different query models, in order to...
achieve a query complexity that is less than the complexity of the actual graph. The query models started from the simple neighbor query, but soon people realized that this was not ideal for minimizing the query complexity for most problems [7, 9]. Therefore, in the case of the minimum cut problem, the cut query was introduced to achieve subquadratic query complexity [13].

There is a vast literature available on the query complexity of problems with classical polynomial time algorithms (Refer to book [8]). However, there has been almost negligible work on algorithmically hard problems [10, 11, 12]. In this paper, we use ideas of parameterized complexity in order to study the query complexity of NP-hard problems. The Hitting Set and Vertex Cover problems are test problems for all new techniques of parameterized complexity and also in every subarea that parameterized complexity has explored. We continue the tradition and study the query complexity of these problems. We start by defining a generalization of a recently introduced query model [2] to handle hypergraphs.

1.1 The model

A hypergraph is a set system \((U(\mathcal{H}),\mathcal{F}(\mathcal{H}))\), where \(U(\mathcal{H})\) is the set of vertices and \(\mathcal{F}(\mathcal{H})\) is the set of hyperedges. A hypergraph \(\mathcal{H}'\) is a sub-hypergraph of \(\mathcal{H}\) if \(U(\mathcal{H}') \subseteq U(\mathcal{H})\) and \(\mathcal{F}(\mathcal{H}') \subseteq \mathcal{F}(\mathcal{H})\). For a hyperedge \(F \in \mathcal{F}(\mathcal{H})\), \(U(F)\) or simply \(F\) denotes the subset of elements that form the hyperedge. A \(d\)-uniform hypergraph has each hyperedge of size \(d\). A packing in a hypergraph \(\mathcal{H}\) is a family \(\mathcal{F}'\) of hyperedges such that for any two hyperedges \(F_1, F_2 \in \mathcal{F}', U(F_1) \cap U(F_2) = \emptyset\).

For us “choose a random hash function \(h : V \rightarrow [N]\)”, means that each vertex in \(V\) is colored with one of the \(N\) colors uniformly and independently at random.

In this paper, for a problem instance \((I,k)\) of a parameterized problem \(\Pi\), a high probability event means that it occurs with probability at least \(1 - \frac{1}{k}\), where \(k\) is the given parameter and \(c\) is a constant. The set \(\{1,2,\ldots,n\}\) is denoted by \([n]\). For a function \(f(k)\), the set of functions \(\mathcal{O}(f(k) \cdot \log k)\), is denoted by \(\tilde{\mathcal{O}}(f(k))\).

Motivated by [2] and [11], we consider the following oracles to look at the parameterized query complexity of NP hard graph problems.

**Generalized \(d\)-partite independent set oracle (GPIS):** For a \(d\)-uniform hypergraph \(\mathcal{H}\), given \(d\) pairwise disjoint non-empty subsets \(A_1,\ldots,A_d \subseteq U(\mathcal{H})\) as input, a GPIS query oracle answers whether there exists an edge \((u_1,\ldots,u_d) \in \mathcal{F}(\mathcal{H})\) such that \(u_i \in A_i\), for each \(i \in [d]\).

**Generalized \(d\)-partite independent set edge oracle (GPISE):** For a \(d\)-uniform hypergraph \(\mathcal{H}\), given \(d\) pairwise disjoint non-empty subsets \(A_1,\ldots,A_d \subseteq U(\mathcal{H})\) as input, a GPISE query oracle outputs a hyperedge \((u_1,\ldots,u_d) \in \mathcal{F}(\mathcal{H})\) such that \(u_i \in A_i\), for each \(i \in [d]\); otherwise, the GPISE oracle reports NULL.

For \(d = 2\), GPIS oracle is same as Bipartite Independent Set (BIS) oracle introduced by Beame et al. [2]. Similarly, we can define BISE oracle as GPISE oracle for \(d = 2\). Notice that BIS is an existence query and is a natural extension of edge existence query. To get a clear motivation behind BIS query, please refer to [2]. BISE (GPISE) is powerful over BIS (GPIS) as BISE (GPISE) can return an edge (a hyperedge) between sets.

As mentioned earlier, queries like degree query, edge existence query, neighbor query, that obtain local information about the graph have its limitation in terms of not being able to achieve efficient query costs [7, 9]. This necessitates looking at powerful queries that
goes beyond obtaining local information and generalizes earlier queries. Beame et al. [2] introduced BIS query model and approximately estimated the number of edges in a graph.

In the context of NP-Hard problems, it is not known if any problem can have efficient query complexity with conventional query models. So, it is reasonable to study query complexity for parameterized versions of NP-Hard problems. Iwama and Yoshida [11] initiated the study of parameterized version of some NP-Hard problems in the graph property testing framework with the access to standard oracles. We will give the details of their work in Section 1.3 and compare with ours. Now, a natural question to ask is “can we improve the query complexity (of NP-Hard problems) with no assumption on the input by considering a relatively stronger oracle” ? As a first step in this direction, we use GPISE (GPIS) oracles, which are nothing but BIS oracle for hypergraphs, to study parameterized decision (optimization) version of Hitting Set. We believe that these query models will be useful to study the (parameterized) query complexity of other NP-Hard problems.

1.2 Problem definition and our results

The $d$-Hitting-Set problem is defined as follows. Note that $d$ is a constant in this paper and $HS(H)$ denote a minimum hitting set of a hypergraph $H$.

**$d$-Hitting-Set**

**Input:** The set of vertices $U(H)$ of a $d$-uniform hypergraph $H$, the access to a GPISE oracle, and a positive integer $k$.

**Output:** A set $HS(H)$ having at most $k$ vertices such that any hyperedge in $H$ intersects with $HS(H)$ if such a set exists. Otherwise, we report such a set does not exist.

The $d$-Decision-Hitting-Set problem is the usual decision version of $d$-Hitting-Set; here the oracle access is to GPIS instead of GPISE.

In our solution framework, we make queries to oracles to build a reduced instance of the problem. On this reduced instance, one can run the traditional (FPT) algorithms. While stating the results, we will bother only about the number of queries required to build the reduced instance. In the query complexity setting, the algorithms are required to make bounded number of queries (good bounds on the total time complexity is not an issue). Our main focus in this paper is to make the query complexity results parameterized, in the sense that they have query complexities bounded by some input parameters of the problem. So, our bounds on the query complexity are not directly comparable with the time complexities of the FPT algorithms in the literature of parameterized complexity. Our results hold with high probability. Our methods use the technique of color coding [1, 5] to restrict the number of queries required to generate a reduced instance of interest. The main result of our paper is the following.

▶ Theorem 1.1. $d$-Hitting-Set can be solved with $\tilde{O}(k^{2d})$ GPISE queries and $d$-Decision-Hitting-Set can be solved with $\tilde{O}(k^{2d^2})$ GPIS queries.

Our solution to $d$-Hitting-Set needs us to solve another problem of interest termed as $d$-Packing in a hypergraph, which is a generalization of Matching in a graph. We describe the sketch of our query procedure to solve $d$-Packing in Section 2. Section 3 has the detailed study on Hitting Set. Table 1 gives the overview of our results.
Table 1 Query complexities for hypergraph problems using GPIS and GPISE oracles. Observe that VERTEX COVER results follow by putting \( d = 2 \) in the above table.

<table>
<thead>
<tr>
<th>Problems</th>
<th>Query Oracles</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIS</td>
<td>GPISE</td>
</tr>
<tr>
<td>( d )-Hitting-Set</td>
<td>( \mathcal{O}(k^{2d}) )</td>
</tr>
<tr>
<td>( d )-Decision-Hitting-Set</td>
<td>( \mathcal{O}(k^{2d}) )</td>
</tr>
<tr>
<td>( d )-Promised-Hitting-Set</td>
<td>( \mathcal{O}(k^{2d}) )</td>
</tr>
<tr>
<td>( d )-Packing</td>
<td>( \mathcal{O}(k^{2d}) )</td>
</tr>
</tbody>
</table>

1.3 Related Works

Several query complexity models have been proposed in the literature to study various problems [7, 9]. The only work prior to ours related to parameterization in the query complexity model was by Iwama and Yoshida [11]. They studied property testing for several parameterized NP optimization problems in the query complexity model. For the query, they could ask for the degree of a vertex, neighbors of a vertex and had an added power of sampling an edge uniformly at random, which is quite unlike in usual query complexity models. To justify the added power of the oracle to sample edges uniformly at random, they have shown that \( \Omega(\sqrt{n}) \) degree and neighbor queries are required to solve VERTEX-COVER. Apart from that, an important assumption in their work is that the algorithms knew the number of edges, which is not what is usually done in query complexity models. Also, the algorithms that are designed gives correct answer only for stable instances. In contrast, our query oracles do not use any randomness, does not know the number of edges, consider all instances, and have a unifying structure. Hence, in this paper, oracles have less power than that of [11] in the context of amount of randomness used by the oracles. Of significance to us, is the vertex cover problem. Their vertex cover algorithm admits a query complexity of \( \mathcal{O}(\frac{k}{\epsilon}) \) and either finds a vertex cover of size at most \( k \) or decides that there is no vertex cover of size bounded by \( k \) even if we delete \( \epsilon m \) edges, where the number of edges \( m \) is known in advance. In contrast, our algorithm uses BISE query for the vertex cover problem; it does not need to estimate the number of edges. Our algorithm admits a query complexity of \( \mathcal{O}(k^3) \) and we either find a vertex cover of size at most \( k \) if it exists or decide that there is no vertex cover of size bounded by \( k \). If it is promised that the vertex cover is bounded by \( k \), then we can give an algorithm that makes \( \mathcal{O}(k^2) \) BISE queries. These results on the VERTEX COVER are not the main focus of this paper and are mentioned in the full version [3]. The main focus of this paper is our results on \( d \)-Hitting-Set; extension of VERTEX COVER to \( d \)-Hitting-Set requires a deeper understanding of stability of sunflowers under random sampling. Hence, it is evident that GPIS (BIS) and GPISE (BISE) open up a new dimension in the study of query complexity. It will be interesting to study what other NP-hard problems can be solved efficiently with these oracles.

Recent papers have considered strengthened query complexity models. In [2], the BIS oracle was introduced to design better edge estimation algorithms. In the same work, the IS oracle was also introduced, to estimate the number of edges, where the input to the oracle is a vertex subset \( A \subseteq V(G) \) and the output is 1, if the subgraph of \( G \) induced by \( A \) is an independent set and 0, otherwise. Similarly, in [13], the cut query was introduced to obtain better query complexity for minimum cut problem.
2 \(d\)-Packing

We first define \(d\)-Packing and then design the query procedure.

\[d\text{-Packing}\]

\textbf{Input:}\ The set of vertices \(U(H)\) of a \(d\)-uniform hypergraph \(H\), the access to a GPISE oracle, and a positive integer \(k\).

\textbf{Output:}\ A pairwise disjoint set of at least \(k\) hyperedges if such a set of hyperedges exists. Otherwise, we report such a set of hyperedges does not exist.

As the usual matching in a graph (denoted here as MATCHING) is a special case of \(d\)-Packing, we explain the main ideas of the query procedure of \(d\)-Packing with MATCHING. In MATCHING, our objective is to either report a matching of at least \(k\) edges or decide there does not exist a matching of size at least \(k\). We use a hash function to color all the vertices of \(G\). In fixing the number of colors needed, we need to ensure that the endpoints of the matched edges belong to different color classes. If the hash function uses \(O(k^2)\) colors, then with constant probability the endpoints of a \(k\)-sized edge set, that certifies the existence of a matching of size at least \(k\), will be in different color classes. For each pair of color classes, we query the BISE oracle and construct a subgraph \(\hat{G}\) according to the outputs of BISE queries. We will show that if \(G\) has a matching of \(k\) edges, then \(\hat{G}\) has a matching of \(k\) edges. As \(\hat{G}\) is a subgraph of \(G\), any matching of \(\hat{G}\) is also a matching of \(G\) and the size of maximum matching in \(\hat{G}\) is less than that of \(G\). So, we report the required answer from the matching of \(\hat{G}\). By repeating the query procedure for \(O(\log k)\) times and taking maximum of all the outcomes, we can report the correct answer with high probability. We carry over the above ideas to the hypergraph setting with the oracle being GPISE. Let \(\text{Pack}(\mathcal{H})\) denote a maximum packing of \(\mathcal{H}\).

\[\text{Theorem 2.1.}\ \text{\(d\)-Packing can be solved with } \tilde{O}(k^{2d}) \text{ GPISE queries.}\]

\[\text{Proof Sketch.}\ \text{Observe that it is enough to give an algorithm that solves \(d\)-Packing with probability at least 2/3 by using } O(k^{2d}) \text{ GPISE queries. The details are in the full version [3].}\]

We choose a random hash function \(h : U(H) \rightarrow [\gamma k^2]\), where \(\gamma = 100d^2\). Let \(U_i = \{u \in U(H) : h(u) = i\}\), where \(i \in [\gamma k^2]\). Note that \(\{U_1, \ldots, U_{\gamma k^2}\}\) form a partition of \(U(H)\), where some of the \(U_i\)'s can be empty. We make a GPISE query with input \((U_{i_1}, \ldots, U_{i_d})\) for each \(1 \leq i_1 < \ldots < i_d \leq \gamma k^2\) such that \(U_{i_j} \neq \emptyset \forall j \in [d]\). Observe that we make \(O(k^{2d})\) queries to the GPISE oracle. Let \(F'\) be the set of hyperedges that are output by the \(O(k^{2d})\) GPISE queries. Now, we can generate a sub-hypergraph \(\tilde{H}\) of \(H\) such that \(U(\tilde{H}) = U(H)\) and \(F(\tilde{H}) = F'\). We find \(\text{Pack}(\tilde{H})\). If \(|\text{Pack}(\tilde{H})| \geq k\), then we report \(\text{Pack}(\tilde{H})\) as \(\text{Pack}(H)\). Otherwise, we report there does not exist a packing of size \(k\). The correctness of our query procedure follows from Lemma 2.2 (proof is in the full version [3]) along with the fact that any packing of \(\tilde{H}\) is also a packing of \(H\), as \(\tilde{H}\) is a sub-hypergraph of \(H\).

\[\text{Lemma 2.2.}\ \text{If } |\text{Pack}(\mathcal{H})| \geq k, \text{ then } |\text{Pack}(\tilde{H})| \geq k \text{ with probability at least 2/3.}\]

3 Algorithm for Hitting Set (Theorem 1.1)

3.1 Our ideas in a nutshell

The main ideas explained with Vertex Cover

The \(d\)-Hitting-Set problem with \(d = 2\) is \(\text{Vertex-Cover}\). We first explain the intuition behind our algorithm for \(d\)-Hitting-Set with \(\text{Vertex-Cover}\). The first step is to solve the problem on instances where there is a promise of a \(\text{Vertex-Cover}\) solution of size
at most $k$. For this promised version, we use a hash function to color all the vertices of the graph $G$. We sample a subgraph of $G$ by querying the BISE oracle for each pair of color classes. We sample several such subgraphs of $G$ using the BISE oracle, and finally take the union of these subgraphs to form a single subgraph $\hat{G}$ of $G$. Finally, we analyse that a minimum vertex cover of $\hat{G}$ is also a minimum vertex cover of $G$ and vice versa. Our analysis is inspired by the analysis of the streaming algorithm for $\text{Vertex-Cover}$ [4], and presented in the full version [3]. The non-promised version of $\text{Vertex-Cover}$ can be solved by using the algorithm for the promised version along with the algorithm explained for $\text{Matching}$ in Section 2. If there exists a matching of size more than $k$, then the vertex cover is also more than $k$. Otherwise, the vertex cover is bounded by $2k$. Now we can use our algorithm for the promised version of $\text{Vertex Cover}$ to find an exact vertex cover from which we can give final answer to the non-promised $\text{Vertex Cover}$. When we consider the decision version of $\text{Vertex-Cover}$, we only need access to the BIS oracle. We use the fact that the $\text{Vertex-Cover}$ problem has an efficient representative set of edges [5] associated with it (please refer to the full version [3]) to solve $\text{Decision-Vertex-Cover}$. This helps us to design an algorithm with access to the BIS oracle. This technique also works for $d$-$\text{Decision-Hitting-Set}$.

Moving from Vertex Cover to $d$-Hitting-Set

The algorithm for $d$-$\text{Hitting-Set}$, having a query complexity of $\tilde{O}(k^{2d})$ GPISE queries, will use an algorithm admitting query complexity $\tilde{O}(k^d)$ for a promised version of this problem. In the promised version, we are guaranteed that the input instance has a hitting set of size at most $k$. The main idea to solve the promised version is to sample a suitable sub-hypergraph having bounded number of hyperedges, using GPISE queries, such that the hitting set of the sampled hypergraph is a hitting set of the original hypergraph and vice versa. We use the stability of sunflowers under random sampling. Recall that a hypergraph can be thought of as a set system. The core of a sunflower is the pairwise intersection of the hyperedges present in the sunflower, which is formally defined as follows.

Definition 3.1. Let $\mathcal{H}$ be a $d$-uniform hypergraph; $S = \{F_1, \ldots, F_t\} \subseteq \mathcal{F}(\mathcal{H})$ is a $t$-sunflower in $\mathcal{H}$ if there exists $C \subseteq U(\mathcal{H})$ such that $F_i \cap F_j = C$ for all $1 \leq i < j \leq t$. $C$ is defined to be the core of the sunflower $S$ in $\mathcal{H}$ and $\mathcal{P} = \{F_i \setminus C : i \in [t]\}$ is defined as the set of petals of the sunflower $S$ in $\mathcal{H}$.

The core of a sunflower can be large, or significant, or small; based on the number of hyperedges forming the sunflower. We define large, significant and small in such a way that each large core is significant and each significant (and thus, large) core must intersect with any hitting set. The formal definition of different types of cores is given below.

Definition 3.2. Let $S_H(C)$ denote the maximum integer $t$ such that $C$ is the core of a $t$-sunflower in $\mathcal{H}$. If $S_H(C) > 10dk$, the core $C$ is said to be large. If $S_H(C) > k$, core $C$ is said to be significant.

The promise that the hitting set is bounded by $k$, will help us (i) to bound the number of hyperedges that do not contain any large core as a subset, (ii) to guarantee that all the large cores, that do not contain any significant cores as subsets in the original hypergraph, are significant in the sampled hypergraph with high probability, and hence will intersect any hitting set of the sampled hypergraph, (iii) to guarantee that all the hyperedges that do not contain any large core as a subset, are present in the sampled hypergraph with high probability. Using the above properties, we can prove that reporting the hitting set of the sampled hypergraph as the hitting set of the original graph is correct with high probability. The formal definitions and arguments are given in Section 3.3.
In this Section we also give algorithm for \(d\)-Decision-Hitting-Set, where we have access to the GPISE oracle and obtain an algorithm with query complexity \(\tilde{O}(kd^e)\). The main idea to solve \(d\)-Decision-Hitting-Set is to use the concept of representative sets \([5]\) (For details see the full version \([3]\)). The size of a \(k\)-representative set corresponding to a hypergraph is bounded by \(O(k^d)\). Thus, the number of vertices that are present in the \(k\)-representative set is also bounded by \(O(dk^d)\). All the \(O(dk^d)\) vertices will be uniquely colored with high probability if enough number of colors are used for the hash function. Then we make GPIS queries to extract a sufficient number of hyperedges such that the hyperedges corresponding to the representative set are embedded in the sampled sub-hypergraph. The formal arguments are given in Section 3.3.

### 3.2 \(d\)-Promised-Hitting-Set

In this part, we study the following problem.

<table>
<thead>
<tr>
<th>(d)-Promised-Hitting-Set</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong></td>
</tr>
<tr>
<td><strong>Output:</strong></td>
</tr>
</tbody>
</table>

For \(d\)-Promised-Hitting-Set, we design an algorithm with query complexity \(\tilde{O}(kd^d)\).

> **Theorem 3.3.** There exists an algorithm that makes \(\tilde{O}(kd^d)\) GPISE queries and solves \(d\)-Promised-Hitting-Set with high probability.

Here, we give an outline of the algorithm. The first step of designing this algorithm involves, for a positive integer \(b\), a sampling primitive \(S_b\) for the problem. Let \(H\) be the \(d\)-uniform hypergraph whose vertex set \(U(H)\) is known and hyperedge set \(F(H)\) is unknown to us. Let \(h : U(H) \rightarrow [b]\) be a random hash function. Let \(U_i = \{ u \in U(H) : h(u) = i\}\), where \(i \in [b]\). Note that \(U_1, \ldots, U_b\) form a partition of \(U(H)\), some of the \(U_i\)’s can be empty. We make a GPISE query with input \((U_{i_1}, \ldots, U_{i_d})\) for each \(1 \leq i_1 < \ldots < i_d \leq b\) such that \(U_{i_j} \neq \emptyset \ \forall j \in [d]\). Observe that we make \(O(b^d)\) queries to the oracle. Let \(F'\) be the set of hyperedges that are output by the \(O(b^d)\) GPISE queries. Now, we can generate a sub-hypergraph \(H'\) of \(H\) such that \(U'(H') = U(H)\) and \(F'(H') = F'\).

Henceforth, the term edge and graph would essentially mean a hyperedge and a \(d\)-uniform hypergraph, respectively.

We find \(\alpha \log k\) samples by calling the sampling primitive \(S_{\beta k}\) for \(\alpha \log k\) times, where \(\alpha = 100kd^2\) and \(\beta = 100kd^2d^{d+5}\). Let the subgraphs resulting from the sampling be \(H_1, \ldots, H_{\alpha \log k}\). Let \(\hat{H} = H_1 \cup \ldots \cup H_{\alpha \log k}\). Note that we can construct \(\hat{H}\) by making \(\tilde{O}(kd^d)\) GPISE queries. Observe that if we prove the following lemma, then we are done with the proof of Theorem 3.3 (the detailed proof is in the full version \([3]\)).

> **Lemma 3.4.** If \(|HS(H)| \leq k\), then \(HS(H) = HS(\hat{H})\) with high probability.

To prove Lemma 3.4, we need some intermediate results. We state the following proposition and then define some sets, which will be needed for our analysis.

> **Proposition 3.1** \([6]\). Let \(H\) be a \(d\)-uniform hypergraph. If \(|F(H)| > d!k^d\), then there exists a \((k + 1)\)-sunflower in \(H\).

> **Definition 3.6.** In the hypergraph \(H\), \(C\) is the set of large cores; \(F_s\) is the family of edges that do not contain any large core; \(C'\) is the family of large cores none of which contain a significant core as a proper subset.
The following two results (Lemma 3.7 and 3.8) give useful bounds with respect to the input instances of $d$-PROMISED-HITTING-SET.

**Lemma 3.7.** If $|HS(\mathcal{H})| \leq k$, then $|\mathcal{F}_s| \leq d!(10dk)^d$.

**Proof.** If $|\mathcal{F}_s| > d!(10dk)^d$, then there exists a $(10dk+1)$-sunflower $\mathcal{S}$ in $\mathcal{H}$ by Proposition 3.5 such that each edge in $\mathcal{S}$ belongs to $\mathcal{F}_s$. First, since $HS(\mathcal{H}) \leq k$, the core $C_\mathcal{H}(\mathcal{S})$ of $\mathcal{S}$ must be non-empty. Note that $C_\mathcal{H}(\mathcal{S})$ is a large core and $C_\mathcal{H}(\mathcal{S})$ is contained in every edge in $\mathcal{S}$. Observe that we arrived at a contradiction, because any edge in $\mathcal{S}$ is also an edge in $\mathcal{F}_s$ and any edge in $\mathcal{F}_s$ does not contain a large core by definition. Hence, $|\mathcal{F}_s| \leq d!(10dk)^d$. □

**Lemma 3.8.** If $|HS(\mathcal{H})| \leq k$, then $|\mathcal{C}'| \leq (d-1)k^{d-1}$.

**Proof.** Let us consider the set system of all cores in $\mathcal{C}'$. Note that the number of elements present in each core in $\mathcal{C}'$ is at most $d-1$. If $|\mathcal{C}'| > (d-1)! \cdot k^{d-1}$, then there exists a $k+1$-sunflower $\mathcal{S}'$. Let $C_1, \ldots, C_{k+1}$ be the sets present in the sunflower $\mathcal{S}'$ and let $C_{\mathcal{S}'}$ be the core of $\mathcal{S}'$. Observe that if $C_{\mathcal{S}'} = C_1 \cap \ldots \cap C_{k+1} = \emptyset$, then $|HS(\mathcal{H})| > k$.

Now consider the following observation for the case when $C_{\mathcal{S}'}$ is non-empty.

**Observation 3.9.** If $C_{\mathcal{S}'}$ is non-empty, then $C_{\mathcal{S}'}$ is the pair-wise intersection of a family of $k+1$ edges in $\mathcal{H}$.

**Proof.** Let $A_i$ be the set of at least $10dk$ edges that form a sunflower with core $C_i$, where $i \in [k+1]$. Observe that this is possible as each $C_i$ is a large core. Before proceeding further, note that $C_i \cap C_j = C_{\mathcal{S}'}$ and $(C_i \setminus C_{\mathcal{S}'}) \cap (C_j \setminus C_{\mathcal{S}'}) = \emptyset$ for all $i, j \in [k+1]$ and $i \neq j$.

Consider $B_i \subseteq A_i$ such that for each $F \in B_i$, $F \cap C_j = C_{\mathcal{S}'} \forall j \neq i$ and $|B_i| \geq 9dk$. First, we argue that $B_i$ exists for each $i \in [k+1]$. Recall that for each $j \in [k+1]$, $|C_j| \leq d-1$. Also, for any pair of edges $F_1, F_2 \in A_i$, $(F_1 \setminus C_i) \cap (F_2 \setminus C_i) = \emptyset$. Thus, using the fact that $C_i \cap C_j = C_{\mathcal{S}'}$ for $i \neq j$, a vertex in $C_i \setminus C_{\mathcal{S}'}$ can belong to at most one edge in $A_i$. This implies that there are at most $(d-1)k < dk$ sets $F$ in $A_i$ such that $F \cap C_j \neq C_{\mathcal{S}'}$ for some $j \neq i \in [k+1]$. We can safely assume that $k+1 \geq d$ and therefore, the number of edges $F \in A_i$ such that $F \cap C_j = C_{\mathcal{S}'} \forall j \neq i \in [k+1]$ is at least $10dk - dk = 9dk$. Next, we argue that there exists $k+1$ edges $F_1, \ldots, F_{k+1}$ such that $F_i \in B_i \forall i \in [k+1]$ and $F_i \cap F_j = C_{\mathcal{S}'}$ for all $i, j \in [k+1]$ and $i \neq j$. We show the existence of the $F_i$’s inductively. For the base case, take any arbitrary edge in $B_1$ as $F_1$. Assume that we have chosen $F_1, \ldots, F_p$, where $1 \leq p \leq k$, such that the required conditions hold. We will show that there exists $F_{p+1} \in B_{p+1}$ such that $F_i \cap F_{p+1} = C_{\mathcal{S}'}$ for each $i \in [p]$. By construction of $B_i$’s, no edge in $B_{p+1}$ intersects with $C_i \setminus C_{\mathcal{S}'}$, $i \leq p$; but every edge in $B_{p+1}$ contains $C_{\mathcal{S}'}$. Also, none of the chosen edges out of $F_1, \ldots, F_p$ intersects $C_i \setminus C_{\mathcal{S}'}$. So, if we can select an edge $F \in B_{p+1}$ such that $F \cap C_{\mathcal{S}'}$ is disjoint from $F_1 \cap C_i \forall i \in [p]$, then we are done. Note that for two edges $F', F'' \in B_{p+1}$, $F' \setminus C_{\mathcal{S}'}$ and $F'' \setminus C_{\mathcal{S}'}$ are disjoint. Consider the set $B'_{p+1} \subseteq B_{p+1}$ such that each edge $F \in B'_{p+1}$ intersects with at least one out of $\{F_1 \setminus C_1, \ldots, F_p \setminus C_p\}$. $|B'_{p+1}| \leq dp \leq dk$, because $(F_1 \setminus C_i) \cap (F_j \setminus C_j) = \emptyset \forall i \neq j \in [p]$ and $|F_i| \leq d, i \in [p]$. As $|B_{p+1}| \geq 9dk$, we select any edge in $B_{p+1}$ \ $B'_{p+1}$ as $F_{p+1}$. □

The above observation implies the following. If $C_{\mathcal{S}'}$ is non-empty, then there exists a $(k+1)$-sunflower in $\mathcal{H}$. So, $S_\mathcal{H}(C_{\mathcal{S}'}) > k$ or equivalently $C_{\mathcal{S}'}$ is a significant core. Note that each $C_i$ contains $C_{\mathcal{S}'}$, which is a significant core; which contradicts the definition of $\mathcal{C}'$. Hence, $|\mathcal{C}'| \leq (d-1)k^{d-1}$.

The following Lemma provides insight into the structure of $\hat{\mathcal{H}}$ and thereby is the most important part of proving Lemma 3.4.
Lemma 3.10. Let $\hat{H} = H_1 \cup \ldots \cup H_{\alpha \log k}$. If $|HS(H)| \leq k$, then (a) $F_s \subseteq F(\hat{H})$, and (b) $\forall C \in C', S_{\hat{H}}(C) > k$ hold with high probability.

Proof Sketch. First, consider the two claims stated below.

Claim 3.11. $\forall i \in [\alpha \log k]$, $P(F \in F(H_i) \mid F \in F_s) \geq \frac{1}{2}$.

Claim 3.12. $\forall i \in [\alpha \log k]$, $P(S_{H_i}(C) > k \mid C \in C') \geq \frac{1}{2}$.

The proofs of Claims 3.11 and 3.12 are involved which we prove below. Observe that Lemma 3.10 follows from Claim 3.11 and 3.12. The detailed proof of Lemma 3.10 is in the full version \[3\].

Proof of Claim 3.11. Without loss of generality, we will prove the statement for the graph $H_1$. Let $h : U(H) \rightarrow [\beta k]$ be the random hash function used in the sampling of $H_1$. Observe that by the construction of $H_1$, $F \in F(H_1)$ if the following two conditions hold:

- $h(u) = h(v)$ if and only if $u = v$, where $u, v \in F$.
- For any $F' \neq F$ and $F' \subseteq F(H)$, $F'$ and $F$ differ in the color of at least one vertex.

Hence, $P(F \notin F(H_1) \mid F \in F_s) \leq \sum_{u,v \in F : u \neq v} P(h(u) = h(v)) + P(\mathcal{E}_1)$, where

$$\mathcal{E}_1 : \exists \text{ an edge } F' \subseteq F(H) \text{ such that } F' \neq F \text{ and } \{h(z) : z \in F\} = \{h(z) : z \in F'\}.$$

Before we bound the probability of the occurrence of $\mathcal{E}_1$, we show the existence of a set $D \subseteq U(H) \setminus F$ of bounded cardinality such that each edge in $F(H) \setminus \{F\}$ intersects with $D$.

Observation 3.13. Let $F \in F_s$. Then there exists a set $D \subseteq U(H) \setminus F$ such that each edge in $F(H) \setminus \{F\}$ intersects with $D$ and $|D| \leq 2^{d+5} d^2 k$.

Proof. For each $C \subseteq F$, consider the hypergraph $H_C$ such that $U(H_C) = U(H) \setminus C$ and $F(H_C) = \{F' : F' \subseteq F(H) \text{ and } F' \cap F = C\}$. First, we prove that the size of $HS(H_C)$ is at most $dS_{\hat{H}}(C)$. For the sake of contradiction, assume that $|HS(H_C)| > dS_{\hat{H}}(C)$. Then we argue that there exists $F' \subseteq F(H_C)$ such that each pair of hyperedges in $F'$ are vertex disjoint and $|F'| > S_{\hat{H}}(C)$. If $|F'| \leq S_{\hat{H}}(C)$, then the vertex set $\{w : w \in F', F' \subseteq F\}$ is a hitting set of $H$, and it has size at most $dS_{\hat{H}}(C)$, which is a contradiction. Therefore, there is a $F' \subseteq F(H_C)$ such that each pair of hyperedges in $F'$ is vertex disjoint and $|F'| > S_{\hat{H}}(C)$. Observe that the set of edges $\{F'' \cup C : F'' \subseteq F\}$ forms a $t$-sunflower in $H$, where $t > S_{\hat{H}}(C)$; which contradicts the definition of $S_{\hat{H}}(C)$.

The required set $D$ is $(HS(H) \setminus F) \cup \bigcup_{C \subseteq F} HS(H_C)$.

If a hyperedge $F^* \in F(H) \setminus \{F\}$ intersects with $F$, then it must intersect with $HS(H_C)$ for some $C \subseteq F$; otherwise $F^*$ intersects with $HS(H) \setminus F$. So, each hyperedge in $F(H) \setminus \{F\}$, intersects with $D$. Now, we bound the size of $D$.

$$|D| \leq |HS(H)| + \left| \bigcup_{C \subseteq F} HS(H_C) \right|$$

$$\leq k + \sum_{C \subseteq F} dS_{\hat{H}}(C) \quad (\because |HS(H)| \leq k \text{ and } |HS(H_C)| \leq dS_{\hat{H}}(C))$$

$$\leq k + 2^d \cdot d \cdot 10dk \quad (\because F \text{ does not contain any large core})$$

$$\leq 2^{d+5} d^2 k$$
With respect to the set $D$, we define another event $E_2$ such that $E_2 \supseteq E_1$

$$E_2 : \exists z \in D \text{ such that } h(z) = h(y) \text{ for some } y \in F.$$ 

We will now bound $\mathbb{P}(E_2)$.

So, $\mathbb{P}(E_2) \leq d \frac{|D|}{|S|} = d \frac{s^d + s^d k}{\beta k} = d \frac{s^d + s^d}{\beta} < \frac{1}{10}$. Putting everything together,

$$\mathbb{P}(F \notin \mathcal{F}(H_1) | F \in F_s) \leq \sum_{u,v \in F ; u \neq v} \mathbb{P}(h(u) = h(v)) + \mathbb{P}(E_2) \leq \frac{d^2}{\beta k} + \mathbb{P}(E_2) \leq \frac{d^2}{\beta k} + \frac{1}{10} < \frac{1}{2}.$$

**Proof of Claim 3.12.** Without loss of generality, we will prove the statement for the graph $H_1$. Let $h : U(H) \to [\beta k]$ be the random hash function used in the sampling of $H_1$.

Let $S$ be the sunflower with core $C$ and $F'$ be the set of edges corresponding to sunflower $S$. Note that $|F'| > 10dk$. Let $F'' \subset F'$ be such that $\forall F \in F'' , (F \setminus C) \cap HS(H) = \emptyset$, and $|F''| = (10d - 1)k$. Note that such an $F''$ exists as $|F''| > 10dk$ and $HS(H) \leq k$.

For $F \in F''$, let $X_F$ be the indicator random variable that takes value 1 if and only if there exists $F' \in F'$ such that $F' \in \mathcal{F}(H_1)$ and $\{h(v) | v \in F'\} = \{h(v) | v \in F\}$. Define $X = \sum_{F \in F''} X_F$. Observe that $S_{H_1}(C)$ is a random variable such that $S_{H_1}(C) \geq X$. Recall that we have to prove $\mathbb{P}(S_{H_1}(C) > k | C \in C') \geq \frac{1}{2}$. So, if we can show $\mathbb{P}(X \leq k) < \frac{1}{2}$, then we are done. Observe that $X_F = 1$ if the following events occur.

$E_1 : h(u) = h(v)$ if and only if $u = v$, where $u,v \in F$.

$E_2 :$ There does not exist $y \in F$ and $z \in HS(H) \setminus C$ such that $h(y) = h(z)$. So, $\mathbb{P}(X_F = 1) \geq \mathbb{P}(E_1 \text{ and } E_2)$ and using the fact that $|HS(H)| \leq k$, we have

$$\mathbb{P}(X_F = 0) \leq \sum_{u,v \in F ; u \neq v} \mathbb{P}(h(u) = h(v)) + \sum_{y \in F} \sum_{z \in HS(H) \setminus \{u\}} \mathbb{P}(h(y) = h(z)) \leq \frac{d^2}{\beta k} + d \cdot \frac{|HS(H)|}{\beta k} < \frac{1}{200}.$$ 

Hence, $\mathbb{E}[X] = \sum_{F \in F''} \mathbb{P}(X_F = 1) \geq (10d - 1)k \cdot \frac{99}{200} > 9dk$.

$$\mathbb{P}(X \leq k) \leq \mathbb{P}\left(|F''| - X \geq (10d - 2)k \implies |F''| = (10d - 1)k\right) \leq \frac{\mathbb{E}[|F''| - X]}{(10d - 2)k} \leq \frac{d - 1}{10d - 2} < \frac{1}{2}.$$ 

The first inequality is by Markov and second one is due to $\mathbb{E}[X] > 9dk$.

Now, we have all the ingredients to prove Lemma 3.4.

**Proof of Lemma 3.4.** First, since $\tilde{H}$ is a subgraph of $H$, a minimum hitting set of $H$ is also a hitting set of $\tilde{H}$. To prove this Lemma, it remains to show that when $|HS(H)| \leq k$, then a minimum hitting set of $\tilde{H}$ is also a hitting set of $H$. By Lemma 3.10, it is true that with high probability $F_s \subseteq F(\tilde{H})$ and $S_{\tilde{H}}(C) > k$ if $C \in C'$. It is enough to show that when $F_s \subseteq F(\tilde{H})$ and $S_{\tilde{H}}(C) > k$, $\forall C \in C'$, then a minimum hitting set of $\tilde{H}$ is also a minimum hitting set of $H$.

First we show that each significant core intersects with $HS(H)$. Suppose there exists a significant core $C$ that does not intersect with $HS(H)$. Let $S$ be a $t$-sunflower in $H$, $t > k$, such that $C$ is the core of $S$. Then each of the $t$ petals of $S$ must intersect with $HS(H)$. 

\begin{align*}
\mathbb{P}(X \leq k) \leq \mathbb{P}\left(|F''| - X \geq (10d - 2)k \implies |F''| = (10d - 1)k\right) \\
\leq \frac{\mathbb{E}[|F''| - X]}{(10d - 2)k} \leq \frac{d - 1}{10d - 2} < \frac{1}{2}.
\end{align*}
But the petals of any sunflower are disjoint. This implies $HS(\mathcal{H}) \geq t > k$, which is a contradiction. So, each significant core intersects with $HS(\mathcal{H})$. As large cores are significant, each large core also intersects with $HS(\mathcal{H})$.

Let us consider a subhypergraph of $\mathcal{H}$, say $\mathcal{H}_1$, with the following definition. Take a large core $C_1$ in $\mathcal{H}$ that contains a significant core $C_2$ as a subset. Let $S_1$ be a sunflower with core $C_1$. Let $S_2$ be a sunflower with core $C_2$ that has more than $k$ petals. Note that there can be at most one hyperedge $F_1$ of $S_1$ that is also present in $S_2$. We delete all hyperedges participating in $S_1$ except $F_1$. The remaining hyperedges remain the same as in $\mathcal{H}$. Notice that a hitting set of $\mathcal{H}_1$ is also a hitting set of $\mathcal{H}$; the significant core $C_2$ remains significant in $\mathcal{H}_1$. Thus, any hitting set of $\mathcal{H}_1$ must intersect with $C$ and therefore, must hit all the hyperedges of $S_1$. We can think of this as a reduction rule, where the input hypergraph and the output hypergraph have the same sized minimum hitting sets. Let $\mathcal{H}$ be a hypergraph obtained after applying the above reduction rule exhaustively on $\mathcal{H}$. The following properties must hold for $\mathcal{H}$: (i) $HS(\mathcal{H}) = HS(\mathcal{H})$, (ii) all large cores in $\mathcal{H}$ do not contain significant cores as subsets, (iii) all hyperedges of $F_x$ in $\mathcal{H}$ are still present in $\mathcal{H}$.

By Lemma 3.10, it is also true with high probability that $S_2(C) > k$ when $C$ is a large core of $\mathcal{H}$ that does not contain any significant core as a subset. Note that the arguments in Lemma 3.10 can be made for such large cores without significant cores in $\mathcal{H}$. Thus, we continue the arguments with the assumption that $S_2(C) > k$ when $C$ is a large core of $\mathcal{H}$ that does not contain any significant core as a subset.

Now we show that when $HS(\mathcal{H}) \leq k$, $HS(\mathcal{H}) = HS(\mathcal{H})$. We know that $F_x \subseteq F(\mathcal{H})$. That is, any hyperedge that does not contain any large core as a subset, is present in $\mathcal{H}$. Each hyperedge in $F_x$ must be covered by any hitting set of $\mathcal{H}$ as well as any hitting set of $\mathcal{H}$ and $\mathcal{H}$. Now, it is enough to argue that an hyperedge $F \in F(\mathcal{H}) \setminus F_x$ must be covered by any hitting set of $\mathcal{H}$. Note that each $F \in F(\mathcal{H}) \setminus F_x$ contains a large core, say $\hat{C}$, which does not contain a significant core as a subset. By our assumption, $\hat{C}$ is a significant core in $\mathcal{H}$ and therefore, must be hit by any hitting set of $\mathcal{H}$.

Putting everything together, when $|HS(\mathcal{H})| \leq k$, each edge in $\mathcal{H}$ is covered by any hitting set of $\mathcal{H}$. Thus, $HS(\mathcal{H}) = HS(\mathcal{H})$.

### 3.3 Algorithms for $d$-Hitting-Set and $d$-Decision-Hitting-Set

Now, we explain the algorithms for $d$-Hitting-Set and $d$-Decision-Hitting-Set.

**Theorem 3.14.** $d$-Hitting-Set can be solved with $\tilde{O}(k^{2d})$ GPISE queries.

**Proof.** Let Pack$(\mathcal{H})$ denote a maximum packing of hypergraph $\mathcal{H}$. By Theorem 2.1, with high probability, we can find Pack$(\mathcal{H})$ if $|\text{Pack}(\mathcal{H})| \geq k + 1$. Otherwise, if $|\text{Pack}(\mathcal{H})| \leq k$, then $|HS(\mathcal{H})| \leq dk$.

If $|\text{Pack}(\mathcal{H})| \geq k + 1$, then $|HS(\mathcal{H})| \geq k + 1$. So, in this case we report that there does not exist any packing of size $k + 1$, by making $\tilde{O}(k^{2d})$ GPISE queries. As $|HS(\mathcal{H})| < dk$, $HS(\mathcal{H})$ can be found using our algorithm for $d$-Promised-Hitting-Set by making $\tilde{O}(k^{2d})$ GPISE queries. If $|HS(\mathcal{H})| \leq k$, with high probability we output $HS(\mathcal{H})$ and if $|HS(\mathcal{H})| > k$, we report there does not exist a hitting set of size at most $k$. The total query complexity is $\tilde{O}(k^{2d})$.

**Theorem 3.15.** $d$-Decision-Hitting-Set can be solved with $\tilde{O}(k^{2d})$ GPISE queries.

**Proof.** Observe that, it is enough to give an algorithm that solves $d$-Decision-Hitting-Set with probability at least $2/3$ by using $O(k^{2d})$ GPIS queries. The details are in the full version [3].
We choose a random hash function $h : U(\mathcal{H}) \to [\gamma k^d]$, where $\gamma = 1009^d d^2$. Let $U_i = \{u \in U(\mathcal{H}) : h(u) = i\}$, where $i \in [\gamma k^d]$. Note that $U_i$'s form a partition of $U(\mathcal{H})$, where some of the $U_i$'s can be empty. We make a GPIS query with input $(U_{i_1}, \ldots, U_{i_d})$ for each $1 \leq i_1 < \ldots < i_d \leq \gamma k^d$ such that $U_{i_j} \neq \emptyset \forall j \in [d]$. Recall that the output of a GPIS query is Yes or No. We create a hypergraph $\hat{\mathcal{H}}$ where we create a vertex for each part $U_i$, $i \in [\gamma k^d]$. We abuse notation and denote $U(\hat{\mathcal{H}}) = \{U_1, \ldots, U_{\gamma k^d}\}$ and $F(\hat{\mathcal{H}}) = \{(U_{i_1}, \ldots, U_{i_d}) : \text{GPIS oracle answers yes when given } (U_{i_1}, \ldots, U_{i_d}) \text{ as input}\}$. Observe that we make $O(k^2 d^2)$ queries to the GPIS oracle. We find $HS(\hat{\mathcal{H}})$ and report $|HS(\mathcal{H})| \leq k$ if and only if $|HS(\hat{\mathcal{H}})| \leq k$. The correctness of our query procedure follows from the following Lemma (proof is in the full version [3]).

**Lemma 3.16.** If $|HS(\hat{\mathcal{H}})| \leq k$, then $|HS(\mathcal{H})| \leq k$ with probability at least $2/3$. ▶

---

**References**

Approximate Minimum-Weight Matching with Outliers Under Translation

Pankaj K. Agarwal\textsuperscript{1}
Department of Computer Science, Duke University, Durham, NC 27708, USA
pankaj@cs.duke.edu

Haim Kaplan
School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel
haimk@tau.ac.il

Geva Kipper
School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel
gevakip@gmail.com

Wolfgang Mulzer\textsuperscript{2}
Institut für Informatik, Freie Universität Berlin, 14195 Berlin, Germany
mulzer@inf.fu-berlin.de

Günter Rote
Institut für Informatik, Freie Universität Berlin, 14195 Berlin, Germany
rote@inf.fu-berlin.de

Micha Sharir\textsuperscript{3}
School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel
michas@tau.ac.il

Allen Xiao\textsuperscript{4}
Department of Computer Science, Duke University, Durham, NC 27708, USA
axiao@cs.duke.edu

Abstract

Our goal is to compare two planar point sets by finding subsets of a given size such that a minimum-weight matching between them has the smallest weight. This can be done by a translation of one set that minimizes the weight of the matching. We give efficient algorithms (a) for finding approximately optimal matchings, when the cost of a matching is the $L_p$-norm of the tuple of Euclidean distances between the pairs of matched points, for any $p \in [1, \infty]$, and (b) for constructing small-size approximate minimization (or matching) diagrams: partitions of the translation space into regions, together with an approximate optimal matching for each region.

2012 ACM Subject Classification Mathematics of computing \rightarrow Combinatorics

\textsuperscript{1} Partially supported by NSF grants CCF-15-13816, CCF-15-46392, IIS-14-08846 and ARO grant W911NF-15-1-0408.

\textsuperscript{2} Partially supported by DFG grant MU/3501/1 and ERC STG 757609.

\textsuperscript{3} Partially supported by ISF Grant 892/13, by the Israeli Centers of Research Excellence (I-CORE) program (Center No. 4/11), by the Blavatnik Research Fund in Computer Science at Tel Aviv University, and by the Hermann Minkowski-MINERVA Center for Geometry at Tel Aviv University.

1 Introduction

The following problem arises in pattern matching: given point sets $A, B$, with $|A| = m$ and $|B| = n$, and $k \leq \min\{m, n\}$, find subsets $A' \subseteq A$ and $B' \subseteq B$ with $|A'| = |B'| = k$ and a transformation $R$ that matches $R(A)$ and $B$ as closely as possible, see Figure 1. We think of $A$ as a collection of features, or interest points of some pattern, that we want to match, bijectively, with similar features in a large image $B$. Moreover, since the coordinate frames for $A$ and $B$ are not necessarily aligned, we want to transform $A$ to get the best possible fit.

This problem comes in many variants, depending on the class of permissible transformations $R$ and on the similarity measure for the match. Here, we want to match $A'$ and $B'$ in a one-to-one manner, where the cost of a matching depends on the distances between matched points. Moreover, we only consider translations as permissible transformations, and write $A + t$ for the set $A$ translated by a vector $t \in \mathbb{R}^2$. A feasible solution is given by a translation $t \in \mathbb{R}^2$ and by a matching $M \subseteq A \times B$ of size $k$ (in short, a $k$-matching): a set of $k$ pairs $(a, b) \in A \times B$ so that any point $a \in A$ or $b \in B$ occurs in at most one pair. The parameter $k$ is part of the input. We consider the $L_p$-cost of such a solution, for some $p \in [1, \infty]$:

$$
\text{cost}_p(M, t) = \text{cost}(M, t) := \begin{cases} 
\left( \frac{1}{k} \sum_{(a, b) \in M} \|a + t - b\|^p \right)^{1/p} & \text{for finite } p, \\
\max_{(a, b) \in M} \|a + t - b\| & \text{for } p = \infty.
\end{cases}
$$

(1)

We will regard $p$ as a fixed constant and will omit it from the notation. Noteworthy special cases arise when $p = 1$ (sum of distances, minimum-weight Euclidean matching), $p = 2$ (root-mean-square matching, in short RMS matching), and $p = \infty$ (bottleneck matching). In (1), we always measure the distances $\|a + t - b\|$ by the Euclidean norm. It is not hard to extend the treatment to other norms, but we stick with Euclidean distances for simplicity.
One important special case occurs when we have a small point set $A$ (the pattern) that we want to locate within a larger set $B$ (the image), and $k = |A| < |B|$. This problem was considered for $p = 2$ by Rote [11] and in subsequent work [3, 8], under the name RMS partial matching. Another important instance has $|A| \approx |B|$ and $k$ slightly smaller than $|A|, |B|$. Now, we want to discard a few outliers from each set, to allow for some erroneous data.

For a fixed translation vector $t \in \mathbb{R}^2$, we define $\text{cost}^*(t) = \min_M \text{cost}(M, t)$ to be the cost of the minimum-cost $k$ matching between $A + t$ and $B$. We set $M_t = \arg \min_M \text{cost}(M, t)$ to be an optimal matching from $A + t$ to $B$, i.e., $\text{cost}^*(t) = \text{cost}(M_t, t)$.

Let $\Pi$ be the set of all $k$-matchings from $A$ into $B$. The function $\text{cost}^*$ is the lower envelope (i.e., the pointwise minimum) of the set of functions $F = \{ t \mapsto \text{cost}(M, t) \mid M \in \Pi \}$. The vertical projection of this lower envelope induces a planar subdivision, called the minimization diagram of $F$. It is denoted by $\mathcal{M} := \mathcal{M}(A, B)$. Each face $\sigma$ of $\mathcal{M}$ is a maximal connected set of points $t$ for which $\text{cost}^*(t)$ is realized by the same matching $M_\sigma$. The combinatorial complexity of $\mathcal{M}$ is the number of its faces. We refer to $\mathcal{M}$ as the $(k)$-matching diagram of $A$ and $B$. We are interested in two questions:

(P1) Compute $t^* = \arg \min_t \text{cost}^*(t)$ and $M^* := M_{t^*}$.

(P2) What is the combinatorial complexity of $\mathcal{M}(A, B)$, and how quickly can it be computed?

These questions have been studied, $p = 2$, by Rote [11] and by Ben-Avraham et al. [3]. Two challenging, still open problems are whether the size of $\mathcal{M}$ is polynomial in both $m$ and $n$, and whether $t^*$ and $M^*$ can be computed in polynomial time. These previous works have raised the questions only for the case $p = 2$, but they are open for arbitrary $p < \infty$. There is extensive work on pattern matching and on computing similarity between two point sets. We refer the reader to [2, 15] for surveys. Here, we confine ourselves to a brief discussion of work directly related to the problem at hand.

Much work has been done on computing a minimum-cost perfect matching in geometric settings. Here, $n = |A| = m = |B| = k$. A minimum-cost perfect matching, for any $L_p$-norm, can be found in $\tilde{O}(n^2)$ time [1, 9, 10]. These algorithms are based on the Hungarian algorithm for a minimum-cost maximum matching in a bipartite graph, and are made more efficient than the general technique by using certain efficient geometric data structures. Thus, they also work when the two point sets $A$ and $B$ have different sizes, say, $|A| = n$ and $|B| = m$, with $k = m \leq n$. In this case, the running time of the algorithm is $\tilde{O}(mn)$.

Approximation algorithms for the minimum-weight perfect matching in geometric settings have been developed in a series of papers; see, e.g., [12] and the references therein. For the case when the weight of a matching is the sum of the Euclidean lengths of its edges, a near-linear algorithm is known [12]. If the weight is the $L_p$-norm of the Euclidean lengths of the edges, for some $p > 1$, then the best known algorithm runs in $\tilde{O}(n^{3/2})$ time [13, 14]. In particular, for RMS matching ($p = 2$) and for $p = 1, \infty$, the time for finding a $(1 + \varepsilon)$-approximate optimal matching is $\tilde{O}(n^{3/2})$, and for a general $p$, the running time is $\tilde{O}(n^{3/2}/\varepsilon^2)$. These algorithms use the scaling method by Gabow and Tarjan [6] that at each scale computes a minimum-weight matching by finding $n$ augmenting paths in $O(\sqrt{n})$ phases, where each phase takes $O(n)$ time (see also [7]). If $|A| = n$, $|B| = m$, and $k = m \leq n$, then the $m$ augmenting paths can be found in $O(\sqrt{m})$ phases, each of which takes $O(n)$ time. Hence, the total running time in this case is $\tilde{O}(\sqrt{mn})$, for $p = 1, 2, \infty$, or $\tilde{O}(\sqrt{mn}/\varepsilon^{3/2})$, for general $p$. When $k \leq m \leq n$, the minimum-weight $k$-matching is constructed, using the geometrically enhanced version.

---

5 The notation $\tilde{O}(\cdot)$ hides polylogarithmic factors in $n$, $m$, and also polylogarithmic factors in $1/\varepsilon$, when we only seek a $(1 + \varepsilon)$-approximate solution.
of the Hungarian algorithm, in \( k \) augmenting steps, each of which can be performed in \( O(n \text{polylog}(n)) \) time. That is, the exact minimum-weight \( k \)-matching can be computed in \( \tilde{O}(kn) \) time. The case of computing an approximate \( k \)-matching is somewhat trickier. If \( k = \Theta(m) \), one can show, adapting the technique in [13], that the running time remains \( O(\sqrt{mn} \text{polylog}(n)) \). For smaller values of \( k \), one can still get a bound depending on \( k \), but we do not treat this case in the paper. It is also much less motivated from the point of view of applications.

Cabello et al. [4] considered optimal shape matching under translations and/or rotations. They considered the more general setting of weighted point sets, where each point of \( A \) and \( B \) comes with a multiplicity or “weight”. Accordingly, the similarity criterion is the earth-mover’s distance, or transportation distance, which measures the minimum amount of work necessary to transport all the weight from \( A \) to \( B \), where transporting a weight \( w \) by distance \( \delta \) costs \( w \cdot \delta \). For the special case of unit weights, this reduces, via the integrality of the minimum-cost flows, to one-to-one matching.

We apply several ideas from Cabello et al.’s paper: (1) the use of point-to-point translations to get constant-factor approximations, (2) the selection of a random subset of these transformations to get fast Monte Carlo algorithms, and (3) tiling the vicinity of these transformations in the parameter space by an \( \varepsilon \)-grid to get \( (1 + \varepsilon) \)-approximations. We go beyond the results of Cabello et al. in the following aspects.

- We give a greedy “disk-eating” algorithm in the space of translations to get an improved deterministic approximation (Theorem 4.5). This idea could be useful for other problems.
- We introduce approximate matching diagrams: Such a diagram is a subdivision of the translation plane together with a matching for each cell. This matching is approximately optimal for every translation in the cell. As a consequence, this diagram provides approximate optimal matchings for all translations. We show that there is an approximate matching diagram of small size, and we describe how to compute it efficiently (Section 2.1).
- Less importantly, our results cover a broader class of similarity measures: The lengths of the \( k \) matching edges can be aggregated in the objective function using any \( \ell_p \)-norm, \( p \geq 1 \), whereas Cabello et al. only dealt with the \( L_1 \) norm. By indentifying the crucial property that lies at the basis of the approximation, namely Lipschitz continuity (Corollary 2.2), this generalization comes without much additional effort. Our results are also slightly more general because we allow outliers (i.e., \( k < \min\{m, n\} \)) whereas Cabello et al. match the smaller set completely.
- By using better data structures, some of our algorithms are more efficient.

We present approximate solutions for (P1) and (P2). They use approximation algorithms for matching between stationary sets as a black box. We write \( W(m, n, k, \varepsilon) \) for the time that is needed to compute a \( (1 + \varepsilon) \)-approximate minimum-weight matching of size \( k \) between two given (stationary) sets \( A \) and \( B \) of \( m \) and \( n \) points in the plane, where the weight is the \( L_p \)-norm of the vector or Euclidean edge lengths, for \( k \leq \min\{m, n\} \) and for a given \( \varepsilon \geq 0 \). We abbreviate \( W(m, n, k) \) as simply \( W(m, n, k) \). Table 1 summarizes the known running times. We obtain two main results:

(i) We present an \( \tilde{O}(mn + \frac{\varepsilon}{\varepsilon} W(m, n, k, \varepsilon/2)) \)-time algorithm for computing a translation vector \( \tilde{t} \) and a \( k \)-matching \( \tilde{M} \) between \( A \) and \( B \) such that \( \text{cost}(\tilde{M}, \tilde{t}) \leq (1 + \varepsilon) \text{cost}^*(t^*) \).

(ii) We present an \( \tilde{O}(mn + \frac{\varepsilon}{\varepsilon} W(m, n, k, \varepsilon/2)) \)-time algorithm for computing a \( (1 + \varepsilon) \)-approximate matching diagram of size \( O\left(\frac{n}{\varepsilon} \log \frac{n}{\varepsilon}\right) \), i.e., a planar subdivision \( \tilde{M} \) and a collection of \( k \)-matchings \( M_\sigma \), one matching for each face \( \sigma \) of \( \tilde{M} \), such that for each face \( \sigma \) of \( \tilde{M} \) and for every \( t \in \sigma \), \( \text{cost}(M_\sigma, t) \leq (1 + \varepsilon) \text{cost}^*(t) \).
The paper is organized as follows. We start with simple solutions to (P1) and (P2) with constant-factor approximations (Section 2). We then refine them to obtain (1+\(\varepsilon\))-approximate solutions, in Section 3. Finally, we present improved algorithms, which attain the bounds claimed in (i) and (ii), in Section 4. All our statements hold for \(p = \infty\). In some cases, the proofs require a special treatment for this case, but for brevity, we will mostly omit the treatment for \(p = \infty\).

## 2 Simple Constant-Factor Approximations

The following lemma establishes a Lipschitz condition for the cost of a matching of size \(k\).

**Lemma 2.1.** Let \(M \subseteq A \times B\) be a matching of size \(k\), and let \(t, \Delta \in \mathbb{R}^2\) be two translation vectors. Then, for any \(p \in [1, \infty]\), the cost under the \(L_p\)-norm satisfies

\[
\text{cost}(M, t + \Delta) \leq \text{cost}(M, t) + \|\Delta\|.
\]

**Proof.** Let \(M = \{(a_1, b_1), \ldots, (a_k, b_k)\}\), and define two nonnegative \(k\)-dimensional vectors \(\vec{v}\) and \(\vec{w}\) by \(\vec{v}_i = \|a_i + t - b_i\|\) and \(\vec{w}_i = \|a_i + t + \Delta - b_i\|\), for \(1 \leq i \leq k\). By the triangle inequality for the Euclidean norm, we have, for each \(i\), \(\vec{v}_i = \|a_i + t + \Delta - b_i\| \leq \|a_i + t - b_i\| + \|\Delta\| = \vec{v}_i + \|\Delta\|\). Thus, we obtain the component-wise inequality \(\vec{w}_i \leq \vec{v}_i + \|\Delta\| \cdot \vec{1}\), where \(\vec{1}\) denotes the \(k\)-dimensional vector in which all components are 1. Now,

\[
\text{cost}(M, t + \Delta) = \frac{\|\vec{w}\|_p}{k^{1/p}} \leq \frac{\|\vec{v} + \|\Delta\| \cdot \vec{1}\|_p}{k^{1/p}} \leq \frac{\|\vec{v}\|_p}{k^{1/p}} + \|\Delta\| \cdot \frac{\|\vec{1}\|_p}{k^{1/p}} = \text{cost}(M, t) + \|\Delta\|,
\]

using the definition (1) of cost, the fact that the \(L_p\)-norm is a monotone function in the components whenever they are nonnegative, and the triangle inequality for the \(L_p\)-norm. \(\blacktriangleleft\)

Here is an immediate corollary of Lemma 2.1:

**Corollary 2.2** (Lipschitz continuity of the optimal cost). For any two translation vectors \(t_1, t_2 \in \mathbb{R}^2\), \(\text{cost}^*(t_2) \leq \text{cost}^*(t_1) + \|t_2 - t_1\|\).

**Proof.** For the respective optimal \(k\)-matchings \(M_1\) and \(M_2\) between \(A + t_1\) and \(B\) and \(A + t_2\) and \(B\),

\[
\text{cost}^*(t_2) = \text{cost}(M_2, t_2) \leq \text{cost}(M_1, t_2) \leq \text{cost}(M_1, t_1) + \|t_2 - t_1\| = \text{cost}^*(t_1) + \|t_2 - t_1\|.
\]

**Approximating \(t^*\) by point-to-point translations.** As in [4], we consider the set \(T = \{-a \mid a \in A, b \in B\}\) of at most \(mn\) point-to-point translations where some point in \(A\) is moved to some point in \(B\). The following simple observation turns out to be very useful:
Lemma 2.3 ([4, Observation 1]). Let $t \in \mathbb{R}^2$ be an arbitrary translation vector, and let $t_0 \in T$ be the nearest neighbor of $t$ in $T$. Then $\text{cost}^*(t) \geq \|t - t_0\|$. 

Proof. By definition, $t_0 = b - a$ is the translation in $T$ with $\|t - t_0\| = \min_{(a',b') \in A \times B} \|t - b' + a'\|$. Thus, for $p < \infty$, all summands in the definition (1) of $\text{cost}^*(t)$ are at least $\|t - t_0\|$, implying $\text{cost}^*(t) \geq \|t - t_0\|$. The last conclusion is trivially valid for $p = \infty$ as well.

Lemma 2.4 ([4, Lemma 1]). There is a translation $t_0 \in T$ with $\text{cost}^*(t_0) \leq 2 \text{cost}^*(t^*)$.

Proof. Let $t^*$ be an optimal translation and $M^*$ a corresponding matching of size $k$. Take the translation $\Delta = b - a - t^* \in \mathbb{R}^2$ for which $\|a + t^* - b\|$ is minimized, over $(a, b) \in M^*$. By Lemma 2.3, $\|\Delta\| \leq \text{cost}^*(t^*)$. The claim now follows from Lipschitz continuity (Corollary 2.2) with $t_1 = t^*$ and $t_2 = t^* + \Delta$, where the latter translation is the desired $t_0 \in T$.

We remark that for RMS matching ($p = 2$), the factor 2 in the lemma can be improved to $\sqrt{2}$. Lemma 2.4 leads to the following simple algorithm for 2-approximating the optimum matching. Compute $T$, and iterate over its elements. For each $t_0 \in T$ compute $\text{cost}^*(t_0)$ (exactly), and return the matching with the minimum weight, in $O(mnW(m, n, k))$ time.

If we are willing to tolerate a slightly larger approximation factor, we can compute, for any $\delta > 0$ and for each $t_0 \in T$, a $(1 + \delta)$-approximate matching, resulting in a $2(1 + \delta)$-approximation. This approach has overall running time $O(mnW(m, n, k, \delta))$.

Theorem 2.5. Let $A, B \subseteq \mathbb{R}^2$, with $|A| = m$ and $|B| = n$, $m \leq n$, and let $k \leq m$ be a size parameter. A translation vector $\hat{t} \in \mathbb{R}^2$ can be computed in $O(mnW(m, n, k))$ time, such that $\text{cost}^*(\hat{t}) \leq 2 \text{cost}^*(t^*)$, where $t^*$ is the optimum translation. Alternatively, for any constant $\delta > 0$, one can compute a translation vector $\hat{t} \in \mathbb{R}^2$ and a $k$-matching $\hat{M}$ between $A$ and $B$, in $O(mnW(m, n, k, \delta))$ time, such that $\text{cost}(\hat{M}, \hat{t}) \leq 2(1 + \delta) \text{cost}^*(t^*)$.

2.1 An Approximate Matching Diagram

We construct a planar subdivision $\tilde{M}$ that approximates the matching diagram $M$ up to factor 3. This means that, for each face $\sigma$ of $\tilde{M}$, there is a single matching $M_{\sigma}$ (that we provide) so that, for each $t \in \sigma$, we have $\text{cost}^*(t) \leq \text{cost}(M_{\sigma}, t) \leq 3 \text{cost}^*(t)$.

We need a lemma that relates the best matching for a given translation $t$ to the closest translation in $T$.

Lemma 2.6. Let $t$ be an arbitrary translation, and let $t_0 \in T$ be its nearest neighbor in $T$, i.e., the translation in $T$ that minimizes the length of $\Delta = t_0 - t$. Then,

$$\text{cost}^*(t) \leq \text{cost}(M_{t_0}, t) \leq 3 \text{cost}^*(t).$$

(Recall that $M_{t_0}$ denotes the optimal matching for $t_0$.)

Proof. Since $M_{t_0}$ is a $k$-matching between $A$ and $B$, we have, by definition, $\text{cost}^*(t) \leq \text{cost}(M_{t_0}, t)$. We prove the second inequality. By Corollary 2.2, $\text{cost}(t_0) \leq \text{cost}^*(t) + \|\Delta\|$, and by Lemma 2.3, $\|\Delta\| \leq \text{cost}^*(t)$. Applying Lemma 2.1, we obtain

$$\text{cost}(M_{t_0}, t) \leq \text{cost}(M_{t_0}, t_0) + \|t - t_0\| = \text{cost}^*(t_0) + \|\Delta\| \leq \text{cost}^*(t) + 2\|\Delta\| \leq \text{cost}^*(t) + 2 \text{cost}^*(t) = 3 \text{cost}^*(t).$$

Our approximate map $\tilde{M}$ is simply the Voronoi diagram $VD(T)$, where each cell $VC(t_0)$, for $t_0 \in T$, is associated with the optimal matching $M_{t_0}$ at $t_0$. Correctness follows immediately from Lemma 2.6. Since the complexity of $VD(T)$ is $O(|T|) = O(mn)$, we have a diagram
of complexity $O(mn)$. For each point $t_0 \in T$, we can either directly compute an optimal k-matching between $A + t_0$ and $B$ and associate the resulting map with $VC(t_0)$, or use the $(1 + \delta)$-approximation algorithm of [13]. In the former case, $VD(T)$ is a 3-approximate matching diagram, and in the latter case it is a $3(1 + \delta)$-approximate matching diagram. We thus conclude the following:

- Theorem 2.7. Let $A, B \subset \mathbb{R}^2$, with $|A| = m$ and $|B| = n$, $m \leq n$, and let $k \leq m$ be a size parameter. There is a 3-approximate k-matching diagram of $A$ and $B$ of size $O(mn)$, and it (and the matchings in each cell) can be computed in $O(mnW(m,n,k))$ time. Alternatively, a $3(1 + \delta)$-approximate matching diagram, for constant $\delta > 0$, of size $O(mn)$ can be computed, using the same planar decomposition, in $O(mnW(m,n,k,\delta))$ time.

For $p = 2$, there is an alternative, potentially better approximating, construction. For each $t \in T$, define the function $f_t(s) := \text{cost}(M_t, s)$, and set $F = \{f_t \mid t \in T\}$. We let $\tilde{M}_0$ be the minimization diagram of the functions in $F$. Simple algebraic manipulations, similar to those for Euclidean Voronoi diagrams, show that $\tilde{M}_0$ is the minimization diagram of a set of $|T| \leq mn$ linear functions, namely, the functions $f_t(s) = 2\sum_{(a,b) \in M_t}(a-b, s) + \sum_{(a,b) \in M_t} \|a-b\|^2$, for $t \in T$. The resulting map $\tilde{M}_0$ is a 3-approximate diagram of complexity $O(mn)$. To see this, consider a Voronoi cell $VC(t_0)$ in $\tilde{M}$. We divide it into subcells in $\tilde{M}_0$, each associated with some matching. All these matchings, other than $M_{t_0}$, have smaller weights than the matching computed for $t_0$, over their respective subcells. Note that this subdivision is only used for the analysis, the algorithm outputs the original minimization diagram. We emphasize that this construction works only for $p = 2$, while the Voronoi diagram applies for any $p \in [1, \infty]$.

For $p = 2$, using the fact that the Euclidean norm is derived from a scalar product, we can improve the constant factors in Lemma 2.4 and Lemma 2.6. However, we chose to present the more general results, since they are simpler and since we derive a more powerful approximation below anyway.

3 Improved Approximation Algorithms

Computing a $(1 + \epsilon)$-approximation of the optimum matching. This algorithm uses the same technique that was used by Cabello et al. [4, Section 4.1, Theorem 6] in a slightly different setting. We include the description of this algorithm as a preparation for the approximate minimization diagram, and for the improved solutions in the following section.

Let $t^*$ be the optimum translation, as above. Our goal is to compute a translation $t$ and a matching $M$ so that $\text{cost}(M, t) \leq (1 + \epsilon) \text{cost}^*(t^*)$.

Suppose we know the translation $t_0 \in T$ that minimizes the length of $\Delta = t_0 - t^*$. By Lemma 2.3 and Lipschitz continuity (Corollary 2.2), $\|\Delta\| \leq \text{cost}^*(t^*) \leq \text{cost}^*(t_0) \leq \text{cost}^*(t^*) + \|\Delta\| \leq 2 \text{cost}^*(t^*)$. Using Theorem 2.5 with $\delta = 1/2$, we compute a 3-approximation for $\text{cost}^*(t)$. This allows us to choose some radius $r_0$ with $2 \text{cost}^*(t^*) \leq r_0 \leq 6 \text{cost}^*(t^*)$. We take the disk $D_0$ of radius $r_0$ centered at $t_0$, and we tile it with the vertices of a square grid of side-length $\delta := \frac{\sqrt{2}}{12} r_0 \leq \frac{\sqrt{2}}{6} \text{cost}^*(t^*)$. We define $G_0$ as the set of vertices of all grid cells that lie in $D_0$ or that overlap $D_0$ at least partially. $G_0$ contains $O(r_0/\delta)^2 = O(1/\epsilon^2)$ vertices.

We compute, by [13], a $(1 + \epsilon/2)$-approximate minimum-weight matching at each translation in $G_0$ and return the one that achieves the smallest weight. Since $t^*$ has distance at most $\delta/\sqrt{2}$ from some grid vertex $g \in G_0$, we have, again by Lipschitz continuity (Corollary 2.2),

$$\text{cost}^*(g) \leq \text{cost}^*(t^*) + \frac{\delta}{\sqrt{2}} \leq \text{cost}^*(t^*) + \frac{\epsilon}{3} \text{cost}^*(t^*) \leq \left(1 + \frac{\epsilon}{3}\right) \text{cost}^*(t^*).$$
Since we compute a \((1 + \varepsilon/2)\)-approximate matching for each grid point, the best computed matching has cost at most \((1 + \varepsilon/3)(1 + \varepsilon/2)\) \(t^*\) \(t^*\) \(t^*\) \(t^*\), assuming \(\varepsilon \leq 1\).

Since we do not know \(t_0\), we apply this procedure to all \(mn\) translations of \(T\), for a total of \(O(mn/\varepsilon^2)\) approximate matching calculations for fixed sets.

**Theorem 3.1.** Let \(A, B \subseteq \mathbb{R}^2\), \(|A| = m \leq |B| = n\), and let \(k \leq m\) be a size parameter and \(0 < \varepsilon \leq 1\) a constant. A translation vector \(t \in \mathbb{R}^2\) and a matching \(\tilde{M}\) of size \(k\) between \(A\) and \(B\) can be computed in \(O\left(\frac{mn}{\varepsilon^2} \cdot W(m, n, k, \frac{\varepsilon}{2})\right)\) time, such that \(\text{cost}(\tilde{M}, t) \leq (1 + \varepsilon)\) \(t^*\).

Cabello et al. [4, Theorem 4] give an \(O\left(\frac{\varepsilon^3 mn}{\varepsilon^2 n}\right)\)-time algorithm for the weighted problem, which includes the matching problem with \(k = m \leq n\) as a special case. It follows the same technique: it solves \(O(mn/\varepsilon^2)\) problems, each with a fixed translation, but each such problem takes longer than in our case because it uses the earth mover’s distance.

**A \((1 + \varepsilon)\)-approximation of \(\tilde{M}\).** We now construct a \((1 + \varepsilon)\)-approximate matching diagram \(\tilde{M}\) of \(A\) and \(B\) by refining \(VD(T)\). Without loss of generality, we assume that \(\varepsilon = 2^{-\alpha}\), for some natural number \(\alpha\), and we set \(u := \log_2(1/\varepsilon) + 2 = \alpha + 2\). We subdivide each Voronoi cell of \(VD(T)\) into smaller subcells, as follows. Fix \(t_0 \in T\). For \(i = 0, \ldots, u\), let \(B_i\) be the square of side-length \(2^i \text{cost}^*(t_0)\), centered at \(t_0\). Set \(B_{-1} = \emptyset\). For \(i = 0, \ldots, u\), we partition \(B_i \setminus B_{i-1}\) into a uniform grid with side-length \(\varepsilon 2^{i-3} \text{cost}^*(t_0)\). We clip each grid cell \(\tau\) to \(\text{VC}(t_0)\), i.e., if \(\tau \cap \text{VC}(t_0) \neq \emptyset\), we take \(\tau \cap \text{VC}(t_0)\) as a face of \(\tilde{M}\). Let \(\tilde{t}\) be the center of the grid cell \(\tau\). We associate \(\tilde{M}_\tau := M_\tau\) with the face \(\tau \cap \text{VC}(t_0)\). Finally, each connected component of \(\text{VC}(t_0) \setminus B_u\) becomes a (possibly non-convex) face of \(\tilde{M}\). There are at most four such faces, and we associate \(M_{\tilde{t}}\) with each of them.

The above procedure partitions \(\text{VC}(t_0)\) into \(O\left(\frac{n}{\varepsilon^2} \log \frac{1}{\varepsilon}\right)\) cells, and their total complexity is \(O(k_0 + \frac{1}{\varepsilon^2} \log \frac{1}{\varepsilon})\), where \(k_0\) is the number of vertices on the boundary of \(\text{VC}(t_0)\). We repeat our procedure for all Voronoi cells of \(VD(T)\). Since the total complexity of \(VD(T)\) is \(O(mn)\), the total complexity of \(\tilde{M}\) is \(O\left(\frac{mn}{\varepsilon^2} \log \frac{1}{\varepsilon}\right)\).

**Lemma 3.2.** \(\tilde{M}\) is a \((1 + \varepsilon)\)-approximate matching diagram of \(A\) and \(B\).

**Proof.** Let \(t \in \mathbb{R}^2\) be an arbitrary translation vector, and let \(t_0 \in T\) be the nearest neighbor of \(t\) in \(T\), i.e., \(t \in \text{VC}(t_0)\). First, consider the case when \(t \not\in B_u\). Then \(\|t - t_0\| \geq 2 \text{cost}^*(t_0)/\varepsilon\) and \(M_{t_0}\) is the matching associated with the cell of \(\tilde{M}\) containing \(t\). Hence, using Lemmas 2.1 and 2.3, we obtain

\[
\text{cost}^*(t) \leq \text{cost}(M_{t_0}, t) \leq \text{cost}^*(t_0) + \|t - t_0\| \leq \left(1 + \frac{\varepsilon}{2}\right)\|t - t_0\| \leq \left(1 + \frac{\varepsilon}{2}\right)\text{cost}^*(t).
\]
Suppose \( t \in B_0 \). Then \( ||t - t_0|| \leq \text{cost}^*(t_0)/\sqrt{2} \). Therefore, by Corollary 2.2,

\[
\text{cost}^*(t) \geq \text{cost}^*(t_0) - ||t - t_0|| \geq \text{cost}^*(t_0) - \frac{1}{\sqrt{2}} \text{cost}^*(t_0) = \left(1 - \frac{1}{\sqrt{2}}\right) \text{cost}^*(t_0).
\]

Let \( \tau \) be the grid cell inside \( B_0 \) containing \( t \), and let \( t_\tau \) be the center of \( \tau \). Then \( ||t - t_\tau|| \leq \varepsilon/\sqrt{2} \text{cost}^*(t_0) \). By Corollary 2.2, \( \text{cost}^*(t_\tau) \leq \text{cost}^*(t) + ||t - t_\tau|| \). Furthermore,

\[
\begin{align*}
\text{cost}(M_\tau, t) &\leq \text{cost}(M_\tau, t_\tau) + ||t - t_\tau|| = \text{cost}^*(t_\tau) + ||t - t_\tau|| \\
&\leq \text{cost}^*(t) + 2||t - t_\tau|| \leq \text{cost}^*(t) + \varepsilon/4\sqrt{2} \text{cost}^*(t_0) \\
&\leq \text{cost}^*(t) + \frac{\varepsilon}{4\sqrt{2}} \cdot \frac{\sqrt{2}}{\sqrt{2} - 1} \text{cost}^*(t) \leq (1 + \varepsilon) \text{cost}^*(t).
\end{align*}
\]

Finally, suppose \( t \in B_i \setminus B_{i-1} \), for some \( i \geq 1 \). Since \( t \notin B_{i-1} \), we have \( ||t - t_0|| \geq 2^{i-2} \text{cost}^*(t_0) \). Let \( \tau \) be the grid cell of \( B_i \setminus B_{i-1} \) containing \( t \), and let \( t_\tau \) be its center. Then \( ||t - t_\tau|| \leq \frac{2^i - 3}{2\varepsilon^2} \cdot \text{cost}^*(t_0) \). Starting with the inequality that was established above, we get

\[
\begin{align*}
\text{cost}(M_\tau, t) &\leq \text{cost}^*(t) + 2||t - t_\tau|| \leq \text{cost}^*(t) + \frac{2^{i-3}\varepsilon}{\sqrt{2}} \text{cost}^*(t_0) \\
&\leq \text{cost}^*(t) + \frac{\varepsilon}{\sqrt{2}} \cdot \frac{2^i - 3}{2\varepsilon^2} \text{cost}^*(t_0) \leq (1 + \varepsilon) \text{cost}^*(t).
\end{align*}
\]

Similar to the \( O(1) \)-approximate matching diagram, we can improve the construction time by setting \( \varepsilon' = \varepsilon/3 \) instead of \( \varepsilon \) and computing a \((1 + \varepsilon/2)\)-approximate optimal matching (instead of the exact matching) for the center of every cell:

\begin{itemize}
\item \textbf{Theorem 3.3. } Let \( A, B \subseteq \mathbb{R}^2 \), with \(|A| = m, |B| = n, m \leq n \) and a size parameter \( k \leq m \).
\item For \( 0 < \varepsilon \leq 1 \), one can compute a \((1 + \varepsilon)\)-approximate \( k \)-matching diagram of \( A \) and \( B \), of size \( O(\frac{mn}{\varepsilon^2} \log \frac{1}{\varepsilon}) \), in \( O(\frac{mn}{\varepsilon^2} \log \frac{1}{\varepsilon})W(m, n, k, \frac{1}{\varepsilon}) \) time.
\end{itemize}

\section{Improved Algorithms}

We now present techniques that improve, by a factor of \( m \) or of \( k \), both algorithms for computing an approximate optimal matching and an approximate matching diagram. These algorithms work well for the case \( k \approx m \), and they deteriorate when \( k \) becomes small. The first algorithm is based on an idea of Cabello et al. [4, Lemma 2]: The best matching contains a substantial number of edges whose length does not exceed the optimum cost by more than a constant factor (cf. Lemma 4.1). This gives a randomized constant-factor approximation algorithm that requires \( O(mn/k) \) approximate matching computations between stationary sets in order to succeed with probability \( \frac{1}{2} \) (Theorem 4.2). We proceed to an improved algorithm that computes a constant-factor approximation with the same number of fixed-translation matching calculations deterministically. By tiling the vicinity of each candidate translation by an \( \varepsilon \)-grid, we then obtain a \((1 + \varepsilon)\)-approximation (Theorem 4.5).

Markov’s inequality bounds the number of items in a sample that are substantially above average. We will use the following consequence of it:

\begin{itemize}
\item \textbf{Lemma 4.1. } Let \( M \) be a matching of size \( k \) between a (possibly translated) set \( A \) and a set \( B \), with cost \( \mu \). Let \( 0 < \varepsilon \leq 1 \). Then the number of pairs \((a, b) \in M\) for which \( ||a - b|| < (1 + \varepsilon)\mu \) is at least \( k - k/(1 + \varepsilon)^p \).
\end{itemize}
Approximate Minimum-Weight Partial Matching Under Translation

Proof. For \( p = \infty \), we interpret \((1 + c)^p\) as \(\infty\), and the result is obvious because \(\|a - b\| < (1 + c)\mu\) for all pairs \((a, b)\). For \(1 \leq p < \infty\), we argue by contradiction. The total number of pairs is \(k\). If there were more than \(k/(1 + c)^p\) pairs \((a, b) \in M\) with \(\|a - b\| \geq (1 + c)\mu\), the total cost would be

\[
\mu = \text{cost}(M) = \left[\frac{1}{k} \sum_{(a, b) \in M} \|a - b\|^p\right]^{1/p} > \left[\frac{1}{k} \cdot k/(1 + c)^p \cdot ((1 + c)\mu)^p\right]^{1/p} = \mu. \quad \blacksquare
\]

Consider the optimal translation \(t^*\) and the corresponding optimal matching \(M^*\). By the lemma, the fraction of the pairs \((a, b) \in M^*\) that satisfy \(\|a + t^* - b\| \leq (1 + c)\text{cost}(t^*)\) is at least \(1 - 1/(1 + c)^p \geq 1 - 1/(e^{c/2})^p = 1 - e^{-cp/2},\) since \(c \leq 1\). Hence, with probability at least \((1 - e^{-cp/2}) \frac{k}{m}\), a randomly chosen \(a \in A\) will participate in such a “close” pair of \(M^*\). We do not know the \(b \in B\) with \((a, b) \in M^*\), so we try all \(n\) possibilities. That is, we choose a single random point \(a_0 \in A\), and we try all \(n\) translations \(b - a_0 \in T\), returning the minimum-weight partial matching over these translations. With probability at least \((1 - e^{-cp/2}) \frac{k}{m}\), we get, by Lemma 2.6, a matching whose weight is at most \(\text{cost}(t^*) + (1 + c)\text{cost}(t^*) = (2 + c)\text{cost}(t^*)\). The runtime of this procedure is \(n \cdot W(m, n, k)\), or \(n \cdot W(m, n, k, \delta)\) if we compute at each of the above translations \(t_0, a \in A\) and construct a \((1 + \delta)\)-approximation to \(\text{cost}(t_0)\). To boost the success probability, we repeat this drawing process \(s\) times and obtain a \((2 + c)(1 + \delta)\)-approximation to the best matching, with probability at least \(1 - (1 - (1 - e^{-cp/2}) \frac{k}{m})^s\). By setting \(c = \delta = \varepsilon/4\), we get the following theorem.

\begin{theorem}
Let \(A, B \subset \mathbb{R}^2\) with \(|A| = m\) and \(|B| = n\), and let \(k \leq m\) and \(s \geq 1\) be parameters. Then, a translation vector \(t \in \mathbb{R}^2\) and a matching \(M\) of size \(k\) between \(A\) and \(B\) can be computed in \(O(sn \cdot W(m, n, k, \varepsilon/4))\) time, such that \(\text{cost}(M, t) \leq (2 + \varepsilon)\text{cost}(t^*)\) with probability at least \(1 - (1 - (1 - e^{-cp/8}) \frac{k}{m})^s\), for any \(\varepsilon\) with \(0 < \varepsilon \leq 1\).
\end{theorem}

If \(ep\) is small, the probability is approximately equal to the simpler expression \(1 - e^{-s \cdot cpk/8m}\).

Cabello et al. [4] proceeded from this result to a \((1 + \varepsilon)\)-approximation by tiling the vicinity of each selected translation with an \(\varepsilon\)-grid [4, Theorem 7]. We will first replace the randomized algorithm by a deterministic one, and apply the \(\varepsilon\)-grid refinement afterwards.

We now describe a deterministic algorithm for approximating \(t^*\) and the corresponding matching \(M^*\). At a high level, the \(mn\) points of \(T\) are partitioned into \(O(mnk/k)\) clusters of size \(\Omega(k)\), and one point, not necessarily from \(T\), is chosen to represent each cluster. We will argue that the point in the resulting set \(X\) of representatives that is nearest to \(t^*\) yields a matching whose value at \(t^*\) is an \(O(1)\)-approximation of \(\text{cost}(t^*)\).

Here is the main idea of how we cluster the points in \(T\) and construct \(X\) in an incremental manner. In step \(i\), we greedily choose the smallest disk \(D_i\) that contains \(k/2\) points of \(T\) (or all of \(T\), if \(|T| \leq k/2\)), add the center of \(D_i\) to \(X\), delete the points of \(D_i \cap T\) from \(T\), and repeat. Carmi et al. [5] have described an efficient algorithm for this clustering problem. It preprocesses \(T\) into a data structure (consisting of three compressed quadtrees) in \(O(mn \log n)\) time, so that in step \(i\), the disk \(D_i\) can be computed in \(\tilde{O}(k^2)\) time and \(D_i \cap T\) can be deleted from the data structure in \(\tilde{O}(k^2)\) time, leading to an \(\tilde{O}(mnk)\)-time algorithm. They also present a faster approximation algorithm for this clustering problem: in step \(i\), instead of computing the smallest enclosing disk \(D_i\), they show that a disk of radius at most twice that of \(D_i\) that still contains \(k/2\) points of \(T\) can be computed in \(\tilde{O}(k)\) time, and that \(D_i \cap T\) can be deleted in \(\tilde{O}(k)\) time, thereby improving the overall running time to \(\tilde{O}(mnk)\).

This approximation algorithm is sufficient for our purpose. We next give a more formal description of our method.

At the beginning of step \(i\), we have a set \(P_i \subseteq T\) and the current set \(X\). Initially, \(P_1 = T\) and \(X = \emptyset\). We preprocess \(P_1\), in \(\tilde{O}(|T|) = \tilde{O}(mn)\) time, into the data structure described by Carmi et al. [5]. We perform the following operations in step \(i\): if \(P_i = \emptyset\), the algorithm...
terminates. If \( 0 < |P_i| \leq k/2 \), we compute the smallest disk \( D_i \) containing \( P_i \). If \( |P_i| > k/2 \), then let \( \rho_i^* \) be the radius of the smallest disk that contains at least \( k/2 \) points of \( P_i \). Using the algorithm in [5], we compute a disk \( D_i \) of radius \( \rho_i \leq 2\rho_i^* \) containing at least \( k/2 \) points of \( P_i \). We add the center \( \xi_i \) of \( D_i \) to \( X \), and we set \( P_{t+1} := P_t \setminus D_i \). We remove \( P_t \cap D_i \) from the data structure, as described in [5]. Let \( \mathcal{D} \) be the set of disks computed by the above procedure. By construction, \( \rho_i^* \leq \rho_{i+1}^* \), \( \rho_i \leq 2\rho_i^* \leq 2\rho_{i+1}^* \leq 2\rho_{i+1} \), and \( |\mathcal{D}| \leq 2|\mathcal{D}_0|/k \).

The following two lemmas establish the correctness of our method.

\[\text{Lemma 4.3.} \quad \text{Let } t \in \mathbb{R}^2 \text{ be a translation vector, and let } \xi_0 \text{ be its nearest neighbor in } X. \text{ Then } \|t - \xi_0\| \leq 3 \cdot 2^{1/p} \text{cost}^*(t).\]

\[\text{Proof.} \quad \text{Let } D \text{ be the disk of radius } 2^{1/p} \text{cost}^*(t) \text{ centered at } t, \text{ and let } S = D \cap T. \text{ By Lemma 4.1 with } 1 + c = 2^{1/p}, \text{ we have } |S| \geq k/2. \text{ Let } D_i \text{ be the first disk chosen by the above procedure that contains a point } t_0 \text{ of } S, \text{ so } S \subseteq P_i. \text{ We must have } \rho_i^* \leq 2^{1/p} \text{cost}^*(t), \text{ because the smallest disk that contains at least } k/2 \text{ points of } P_i \text{ is not larger than } D. \text{ Hence, } \rho_i \leq 2^{1/p} \text{cost}^*(t), \text{ and}
\]
\[
\|t - \xi_i\| \leq \|t - t_0\| + \|t_0 - \xi_i\| \leq 2^{1/p} \text{cost}^*(t) + \rho_i \\
\leq 2^{1/p} \text{cost}^*(t) + 2 \cdot 2^{1/p} \text{cost}^*(t) = 3 \cdot 2^{1/p} \text{cost}^*(t). \]

\[\text{Lemma 4.4.} \quad \min_{\xi \in X} \text{cost}^*(\xi) \leq (1 + 3 \cdot 2^{1/p}) \text{cost}^*(t^*).\]

\[\text{Proof.} \quad \text{Let } \xi_0 \text{ be the nearest neighbor to } t^* \text{ in } X. \text{ Applying Lemma 4.3 with } t = t^*, \text{ we obtain } \|t^* - \xi_0\| \leq 3 \cdot 2^{1/p} \text{cost}^*(t^*). \text{ By Corollary 2.2, we then have } \text{cost}^*(\xi_0) \leq \text{cost}^*(t^*) + \|t^* - \xi_0\| \leq (1 + 3 \cdot 2^{1/p}) \text{cost}^*(t^*). \]

We fix a constant \( \delta \in (0, 1] \). We compute a \((1 + \delta)\)-approximate \( k \)-matching \( M_\delta \) between \( A + \xi \) and \( B \), for every \( \xi \in X \), and choose the best among them. This will give an \( O(1) \)-approximation of the minimum-cost \( k \)-matching under translation. We can extend this algorithm to yield a \((1 + \varepsilon)\)-approximation algorithm following the same procedure as in Section 3: We draw a disk of radius \((1 + 3 \cdot 2^{1/p} + 4\varepsilon) \text{cost}^*(t^*)\) around each point of \( X \). We draw a uniform grid of cell size \( O(\varepsilon) \) and look at all vertices \( t \) of grid cells that overlap one of these disks at least partially. We compute a \((1 + \varepsilon/2)\)-approximation for the best matching of size \( k \) between \( A + t \) and \( B \) for each of the grid point \( t \) under consideration, and we choose the best matching among them. Putting everything together, we obtain the following:

\[\text{Theorem 4.5.} \quad \text{Let } A, B \subset \mathbb{R}^2, \text{ with } |A| = m \text{ and } |B| = n, \text{ and let } 0 < \varepsilon \leq 1 \text{ and } k \leq \min\{m, n\} \text{ be parameters. Then, a translation vector } t \in \mathbb{R}^2 \text{ and a matching } \tilde{M} \text{ of size } k \text{ between } A \text{ and } B \text{ can be computed in } \tilde{O}(mn + \frac{mn}{\varepsilon^2} W(m, n, k, \frac{\varepsilon}{2})) \text{ time, such that } \text{cost}(\tilde{M}, \tilde{t}) \leq (1 + \varepsilon) \text{cost}^*(t^*).\]

We show that \( \text{VD}(X) \) is indeed an \( O(1) \)-approximate matching diagram of \( A \) and \( B \). This is analogous to Section 2.1 (Lemma 2.6).

\[\text{Lemma 4.6.} \quad \text{Let } t \in \mathbb{R}^2 \text{ be a translation vector, and let } \xi_0 \text{ be its nearest neighbor in } X. \text{ Then, } \text{cost}^*(t) \leq \text{cost}(M_{\xi_0}, t) \leq (1 + 6 \cdot 2^{1/p}) \text{cost}^*(t).\]

\[\text{Proof.} \quad \text{Since } M_{\xi_0} \text{ is a matching of size } k \text{ between } A \text{ and } B, \text{ we have, by definition, } \text{cost}^*(t) \leq \text{cost}(M_{\xi_0}, t). \text{ We now prove the second inequality. By Corollary 2.2, } \text{cost}^*(\xi_0) \leq \text{cost}^*(t) + \|t - \xi_0\|, \text{ Lemma 2.1, and Lemma 4.3,}
\]
\[
\text{cost}(M_{\xi_0}, t) \leq \text{cost}(M_{\xi_0}, \xi_0) + \|t - \xi_0\| = \text{cost}^*(\xi_0) + \|t - \xi_0\| \leq \text{cost}^*(t) + 2\|t - \xi_0\| \leq (1 + 6 \cdot 2^{1/p}) \text{cost}^*(t). \]

\[\text{\textbf{\textmd{\textbf{IIAAC 2018}}}}\]
The combinatorial complexity of $\text{VD}(X)$ is $O(mn/k)$. We can now construct a $(1 + \varepsilon)$-approximate matching diagram by refining each Voronoi cell of $\text{VD}(X)$, as in Section 3, but the constants have to be chosen differently. The diagram has $O\left(\frac{mn}{k\varepsilon^2} \log \frac{1}{\varepsilon}\right)$ cells, and we need $W(m, n, k, \frac{\varepsilon}{2})$ time per cell. We obtain the following:

**Theorem 4.7.** Let $A, B \subset \mathbb{R}^2$, $|A| = m \leq |B| = n$, and let $k \leq m$, $\varepsilon \in (0, 1]$ be parameters. There exists a $(1 + \varepsilon)$-approximate $k$-matching diagram of $A$ and $B$ of size $O\left(\frac{mn}{k\varepsilon^2} \log \frac{1}{\varepsilon}\right)$, and it can be computed in $\tilde{O}(mn) + O\left(\frac{mn}{k\varepsilon^2} \log \frac{1}{\varepsilon} W(m, n, k, \frac{\varepsilon}{2})\right)$ time.

For the case when $cm \leq k \leq (1 - c)n$ for some constant $c > 0$, we can show that the bound in Theorem 4.7 on the size of the diagram is tight in the worst case in terms of $m$, $n$, and $k$ (but not of $\varepsilon$): If $A$ is a unit grid of size $\sqrt{m} \times \sqrt{m}$ and $B$ is a unit grid of size $\sqrt{n} \times \sqrt{n}$, then there are $\Omega(n)$ translation vectors at which $A$ and $B$ are perfectly aligned and have at least $k$ points in common. Thus, any $O(1)$-approximate matching diagram of $A$ and $B$ needs to have $\Omega(n)$ distinct faces.

**References**


New and Improved Algorithms for Unordered Tree Inclusion

Tatsuya Akutsu
Bioinformatics Center, Institute for Chemical Research, Kyoto University
Kyoto 611-0011, Japan

Jesper Jansson
Department of Computing, The Hong Kong Polytechnic University
Hung Hom, Kowloon, Hong Kong, China

Ruiming Li
Bioinformatics Center, Institute for Chemical Research, Kyoto University
Kyoto 611-0011, Japan

Atsuhiro Takasu
National Institute of Informatics
Chiyoda-ku, Tokyo, 101-8430, Japan

Takeyuki Tamura
Bioinformatics Center, Institute for Chemical Research, Kyoto University
Kyoto 611-0011, Japan

Abstract

The tree inclusion problem is, given two node-labeled trees $P$ and $T$ (the “pattern tree” and the “text tree”), to locate every minimal subtree in $T$ (if any) that can be obtained by applying a sequence of node insertion operations to $P$. Although the ordered tree inclusion problem is solvable in polynomial time, the unordered tree inclusion problem is NP-hard. The currently fastest algorithm for the latter is from 1995 and runs in $O(poly(m,n) \cdot 2^{2d}) = O^*(2^{2d})$ time, where $m$ and $n$ are the sizes of the pattern and text trees, respectively, and $d$ is the maximum outdegree of the pattern tree. Here, we develop a new algorithm that improves the exponent $2d$ to $d$ by considering a particular type of ancestor-descendant relationships and applying dynamic programming, thus reducing the time complexity to $O^*(2^d)$. We then study restricted variants of the unordered tree inclusion problem where the number of occurrences of different node labels and/or the input trees’ heights are bounded. We show that although the problem remains NP-hard in many such cases, it can be solved in polynomial time for $c = 2$ and in $O^*(1.8^d)$ time for $c = 3$ if the leaves of $P$ are distinctly labeled and each label occurs at most $c$ times in $T$. We also present a randomized $O^*(1.883^d)$-time algorithm for the case that the heights of $P$ and $T$ are one and two, respectively.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases parameterized algorithms, tree inclusion, unordered trees, dynamic programming

Digital Object Identifier 10.4230/LIPICS.ISAAC.2018.27

1 JSPS KAKENHI #18H04113
2 JSPS KAKENHI #25730005

© Tatsuya Akutsu, Jesper Jansson, Ruiming Li, Atsuhiro Takasu, and Takeyuki Tamura, licensed under Creative Commons License CC-BY
1 Introduction

Tree pattern matching and measuring the similarity of trees are classic problem areas in theoretical computer science. One intuitive and extensively studied measure of the similarity between two rooted, node-labeled trees $T_1$ and $T_2$ is the tree edit distance, defined as the length of a shortest sequence of node insertion, node deletion, and node relabeling operations that transforms $T_1$ into $T_2$ [7]. When $T_1$ and $T_2$ are ordered trees, the tree edit distance can be computed in polynomial time. The first algorithm to achieve this bound ran in $O(n^6)$ time [20], where $n$ is the total number of nodes in $T_1$ and $T_2$, and it was gradually improved upon until Demaine et al. [12] presented an $O(n^3)$-time algorithm thirty years later which was proved to be worst-case optimal under a conjecture that there is no truly subcubic time algorithm for the all pairs shortest paths problem [9]. On the other hand, the tree edit distance problem is NP-hard for unordered trees [25]. It is MAX SNP-hard even for binary trees in the unordered case [24], which implies that it is unlikely to admit a polynomial-time approximation scheme. Akutsu et al. [3, 5] have developed efficient exponential-time algorithms for this problem variant. As for parameterized algorithms, Shasha et al. [19] developed an $O(4^\ell_1+\ell_2 \min(\ell_1, \ell_2)^mn)$-time algorithm for the problem, where $\ell_1$ and $\ell_2$ are the numbers of leaves in $T_1$ and $T_2$, respectively. Using another parameter $k$, an $O^*(2.62k^k)$-time algorithm was developed for the unit-cost edit operation model [4], where $k$ is the edit distance and $O^*(f(\cdots))$ means $O(f(\cdots)poly(m,n))$. See [7] for other related results.

An important special case of the tree edit distance problem known as the tree inclusion problem is obtained when only node insertion operations are allowed. This problem has applications to structured text databases and natural language processing [8, 14, 21]. Here, we assume the following formulation of the problem: given a “text tree” $T$ and a “pattern tree” $P$, locate every minimal subtree in $T$ (if any) that can be obtained by applying a sequence of node insertion operations to $P$. (Equivalently, one may define the tree inclusion problem so that only node deletion operations on $T$ are allowed.) For unordered trees, Kilpeläinen and Mannila [14] proved the problem to be NP-hard in general but solvable in polynomial time when the degree (outdegree) of the pattern tree is bounded from above by a constant. More precisely, the running time of their algorithm is $O(d \cdot 2^{2d \cdot mn})$ time, where $m = |P|$, $n = |T|$, and $d$ is the maximum degree of $P$. Bille and Gørtz [8] gave a fast algorithm for the case of ordered trees, and Valiente [21] developed a polynomial-time algorithm for a constrained version of the unordered case. Also note that the special case of the tree inclusion problem where node insertion operations are only allowed to insert new leaves corresponds to a subtree isomorphism problem, which can be solved in polynomial time for unordered trees [17].

1.1 Practical applications

Due to the rapid advance of AI technology, matching methods for knowledge base become more important. As a fundamental technique for searching knowledge base, researchers in database community have been studying the subtree similarity search. For example, Cohen and Or proposed a subtree similarity search algorithm for various distance functions [11], while Chang et al. proposed a top-k tree matching algorithm [10]. In the Natural Language Processing (NLP) field, researchers are incorporating the deep learning techniques into NLP problems and developing parsing/dependency trees processing algorithms [16]. Bibliographic matching is one of the most popular applications of real-world matching problems [15]. In most cases, single article has at most two or three versions, and it is very rare that single article includes the same name co-authors. Therefore, it may be reasonable to assume that the leaves of $P$ are distinctly labeled and each label occurs at most $c$ times in $T$. 
Table 1. The computational complexity of some special cases of the unordered tree inclusion problem, where the last one is a randomized one. For any tree \( T \), \( h(T) \) denotes the height of \( T \) and \( \text{OCC}(T) \) the maximum number of times that any leaf label occurs in \( T \). As indicated in the table, either all nodes or only the leaves are labeled (the former is harder since it generalizes the latter).

<table>
<thead>
<tr>
<th>Restriction</th>
<th>Labels on</th>
<th>Complexity</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h(T) = 2, h(P) = 1, \text{OCC}(T) = 3, \text{OCC}(P) = 1 )</td>
<td>all nodes</td>
<td>NP-hard</td>
<td>Corollary 8</td>
</tr>
<tr>
<td>( h(T) = 2, h(P) = 2, \text{OCC}(T) = 3, \text{OCC}(P) = 1 )</td>
<td>leaves</td>
<td>NP-hard</td>
<td>Theorem 9</td>
</tr>
<tr>
<td>( \text{OCC}(T) = 2, \text{OCC}(P) = 1 )</td>
<td>all nodes</td>
<td>( \mathcal{O}^{*}(1.8^d) ) time</td>
<td>Theorem 11</td>
</tr>
<tr>
<td>( \text{OCC}(T) = 3, \text{OCC}(P) = 1 )</td>
<td>all nodes</td>
<td>( \mathcal{O}^{*}(1.883^d) ) time</td>
<td>Theorem 12</td>
</tr>
<tr>
<td>( h(T) = 2, h(P) = 1 )</td>
<td>all nodes</td>
<td>( \mathcal{O}^{*}(1.883^d) ) time</td>
<td>Theorem 14</td>
</tr>
</tbody>
</table>

The extended tree inclusion problem was proposed in [18], which is an optimization problem designed to make the unordered tree inclusion problem more useful for practical tree pattern matching applications, e.g., involving glycan data from the KEGG database [13], weblogs data [23], and bibliographical data from ACM, DBLP, and Google Scholar [15]. This problem asks for an optimal connected subgraph of \( T \) (if any) that can be obtained by performing node insertion operations as well as node relabeling operations to \( P \) while allowing non-uniform costs to be assigned to the different node operations; it was shown in [18] that the unrooted version can be solved in \( \mathcal{O}^{*}(2^{2d}) \) time and a further extension of the problem that also allows at most \( k \) node deletion operations can be solved in \( \mathcal{O}^{*}((ed)^k k^{1/2}2^{(dk+4d-k)}) \) time where \( e \) is the base of the natural logarithm.

1.2 New results

We improve the exponential contribution to the time complexity of the fastest known algorithm for the unordered tree inclusion problem (Kilpeläinen and Mannila’s algorithm from 1995 [14]) from \( 2^{2d} \) to \( 2^d \), where \( d \) is the maximum degree of the pattern tree, so that the time complexity becomes \( O(d2^d mn^2) = \mathcal{O}^{*}(2^d) \). This improved bound is achieved by introducing a simple but quite useful idea of minimal inclusion and a different way of dynamic programming. Next, we study the problem’s computational complexity for several restricted cases (see Table 1 for a summary) and give a polynomial-time algorithm for when the leaves in \( P \) are distinctly labeled and every label appears at most twice in \( T \). Then, we derive an \( \mathcal{O}^{*}(1.8^d) \)-time algorithm for the NP-hard case where the leaves in \( P \) are distinctly labeled and each label appears at most three times in \( T \). Both are obtained by effectively utilizing a polynomial-time algorithm for 2-SAT. Finally, we derive a randomized \( \mathcal{O}^{*}(1.883^d) \) time algorithm for the case where the heights of \( P \) and \( T \) are one and two, respectively. It is obtained by a simple but non-trivial combination of the \( \mathcal{O}^{*}(2^d) \) time algorithm, an \( \mathcal{O}^{*}(1.234^m) \) time algorithm for SAT with \( m \) clauses [22], and color-coding [6]. Because of the page limit, some proofs are omitted in this version.

2 Preliminaries

From here on, all trees are rooted, unordered, and node-labeled. Let \( T \) be a tree. A node insertion operation on \( T \) is an operation that creates a new node \( v \) having any label and then: (i) attaches \( v \) as a child of some node \( u \) currently in \( T \) and makes \( v \) become the parent of a (possibly empty) subset of the children of \( u \); or (ii) makes the current root of \( T \) become
algorithms for unordered tree inclusion

A child of \( v \) and lets \( v \) become the new root. For any two trees \( T_1 \) and \( T_2 \), we say that \( T_1 \) is included in \( T_2 \) if there exists a sequence of node insertion operations such that applying the sequence to \( T_1 \) yields \( T_2 \) (i.e., \( T_1 \) is obtained by node deletions from \( T_2 \)).

For a tree \( T \), \( v(T), h(T) \), and \( V(T) \) denote its root, height, and the set of nodes in \( T \), respectively. A mapping between two trees \( T_1 \) and \( T_2 \) is a subset \( M \subseteq V(T_1) \times V(T_2) \) such that for every \( (u_1, v_1), (u_2, v_2) \in M \), it holds that: (i) \( u_1 = u_2 \) if and only if \( v_1 = v_2 \); and (ii) \( v_1 \) is an ancestor of \( u_2 \) if and only if \( v_1 \) is an ancestor of \( v_2 \). \( T_1 \) is included in \( T_2 \) if and only if there is a mapping \( M \) between \( T_1 \) and \( T_2 \) such that \( |M| = |V(T_1)| \) and \( u \) and \( v \) have the same node label for every \( (u, v) \in M \) [20]. Such a mapping is called an inclusion mapping.

In the tree inclusion problem, the input consists of two trees \( P \) and \( T \) (also referred to as the “pattern tree” and the “text tree”), and the objective is to locate every minimal subtree of \( T \) that includes \( P \). Define \( m = |V(P)| \) and \( n = |V(T)| \), and \( d \) denote the maximum degree of \( P \). For any node \( v \), let \( \ell(v) \) and \( Chd(v) \) denote its label and the set of its children. Also let \( Anc(v) \) and \( Des(v) \) denote the sets of strict ancestors and exact descendants of \( v \), respectively, i.e., where \( v \) itself is excluded from these sets. For a node \( v \) in a tree \( T \), \( T(v) \) is the subtree of \( T \) induced by \( Des(v) \). We write \( P(u) \subseteq T(v) \) if \( P(u) \) is included in \( T(v) \) under the condition that \( u \) is mapped to \( v \). For two trees \( T_1 \) and \( T_2 \), \( T_1 \sim T_2 \) denotes that \( T_1 \) is isomorphic to \( T_2 \) (with label information). The following concept plays a key role in our algorithm.

**Definition 1.** We say that \( T(v) \) minimally includes \( P(u) \) (denoted as \( P(u) \prec T(v) \)) if \( P(u) \subseteq T(v) \) holds and there is no \( v' \in Des(v) \) such that \( P(u) \subseteq T(v') \).

**Proposition 2.** Let \( Chd(u) = \{u_1, \ldots, u_d\} \). \( P(u) \subseteq T(v) \) holds if and only if the following conditions are satisfied.

1. \( \ell(u) = \ell(v) \).
2. \( v \) has a set of descendants \( D(v) = \{v_1, \ldots, v_d\} \) such that \( v_i \not\in Des(v_j) \) for all \( i \neq j \).
3. There exists a bijection \( \phi \) from \( Chd(u) \) to \( D(v) \) such that \( P(u_i) \prec T(\phi(u_i)) \) holds for all \( u_i \in Chd(u) \).

**Proof.** Conditions (1) and (2) are obvious. To prove (3), suppose there exists a bijection \( \phi' \) from \( Chd(u) \) to \( D(v) \) such that \( P(u_j) \subseteq T(\phi'(u_j)) \) holds for all \( u_j \in Chd(u) \) and \( P(u_i) \prec T(\phi(u_i)) \) does not hold for some \( u_i \in Chd(u) \). Then, there must exist \( v' \in Des(\phi'(u_i)) \) such that \( P(u_i) \prec T(v') \) holds. Let \( \phi'' \) be the bijection obtained by replacing a mapping from \( u_i \) to \( \phi'(u_i) \) with that from \( u_i \) to \( \phi'(u_i) \). Clearly, \( \phi'' \) gives a part of an inclusion mapping. Repeatedly applying this procedure, we can obtain a bijection satisfying all conditions.

Note that the conditions of this proposition mainly state that all children of \( u \) must be mapped to descendants of \( v \) that do not have ancestor-descendant relationships. Since \( P \) is included in \( T \) if and only if there exists \( v \in V(T) \) such that \( P \prec T(v) \), we focus on how to decide if \( P(u) \prec T(v) \) assuming that whether \( P(u_j) \prec T(v_i) \) holds is known for all \( (u_j, v_i) \) with \( u_j \in Des(u) \cup \{u\}, v_i \in Des(v) \cup \{v\}, (u_j, v_i) \neq (u, v) \).

**Proposition 3.** Suppose that \( P(u) \prec T(v) \) can be decided in \( O(f(d, m, n)) \) time assuming that whether \( P(u_j) \prec T(v_i) \) holds is known for all descendant pairs \( (u_j, v_i) \). Then the unordered tree inclusion problem can be solved in \( O(f(d, m, n)mn) \) time by using a bottom-up dynamic programming procedure.
3 An $O(d2^d mn^2)$-time algorithm

The crucial parts of the algorithm in [14] are the definition of $S(v)$ and its computation (see [14] for the details since our algorithms are significantly different from theirs). For each fixed $u$ in $P$, $S(v)$ is defined by

$$S(v) = \{ U \subseteq \text{Chd}(u) \mid P(U) \subset T(v) \},$$

where $P(U)$ is the forest induced by nodes in $U$ and their descendants and $P(U) \subset T(v)$ means that forest $P(U)$ is included in $T(v)$ (i.e., $T(v)$ can be obtained from $P(U)$ by node insertion operations). Clearly, the size of $S(v)$ is no greater than $2^d$. Note that in this paper, we use $S$ or $S(v)$ only to denote a set, not to denote a subtree. In the algorithm of [14], the following operation is performed from left to right for the children $v_1, \ldots, v_l$ of $v$:

$$S := \{ U \cup R \mid U, R \in S(v_i) \},$$

beginning from $S = \emptyset$, and $S(v)$ is determined based on the resulting $S$. However, this update operation on $S$ causes an $O(d2^d)$ factor because it examines $O(2^d) \times O(2^d)$ set pairs. Therefore, in order to avoid this kind of operation, we need a new approach for computing $S(v)$, as explained below.

Given an unordered tree $T$, we fix any left-to-right ordering of its nodes (the ordering does not affect the correctness). Then, for any two nodes $v_i, v_j \in V(T)$ that do not have any ancestor-descendant relationship, either “$v_i$ is left of $v_j$” or “$v_i$ is right of $v_j$” is uniquely determined. We denote “$v_i$ is left of $v_j$” by $v_i \triangleleft v_j$.

We focus on deciding if $P(u) \triangleleft T(v)$ holds for fixed $(u, v)$ because this part is crucial to reduce the exponential factor (we analyze the whole time complexity in Theorem 7). Assume w.l.o.g. (without loss of generality) that $\text{Chd}(u) = \{ u_1, \ldots, u_d \}$ (i.e., $u$ has $d$ children). For simplicity, we assume until the end of this section that $P(u_i) \sim P(u_j)$ does not hold for any $u_i \neq u_j \in \text{Chd}(u)$. For any $v_i \in V(T(v))$, define $M(v_i)$ by $M(v_i) = \{ u_j \in \text{Chd}(u) \mid P(u_j) \sim T(v_i) \}$. For example, $M(v_0) = \emptyset$, $M(v_2) = \{ u_C \}$, and $M(v_3) = \{ u_D, u_E \}$ in Figure 1. For any $v_i \in V(T(v))$, $LF(v_i, v_j)$ denotes the sets of nodes in $V(T(v))$ each of which is left of $v_i$ (see Figure 1 for an example). Then, we define $S(v, v_i)$ by

$$S(v, v_i) = \{ U \subseteq \text{Chd}(u) \mid P(U) \subset T(LF(v, v_i)) \}$$

$$\cup \{ U \subseteq \text{Chd}(u) \mid (U = U' \cup \{ u_j \}) \wedge (P(U') \subset T(LF(v, v_i))) \wedge (u_j \in M(v_i)) \}$$

where $T(LF(v, v_i))$ is the forest induced by nodes in $LF(v, v_i)$ and their descendants. Note that $P(\emptyset) \subset T(\ldots)$ always holds. The definition of $S(v, v_i)$ leads to a dynamic programming procedure for its computation. We explain $S(v, v_i)$ and related concepts using an example in Figure 1. Suppose that we have the relations of $P(u_A) \sim T(v_1), P(u_B) \sim T(v_1), P(u_C) \sim T(v_2), P(u_D) \sim T(v_3), P(u_E) \sim T(v_3), P(u_D) \sim T(v_3), P(u_E) \sim T(v_3)$. Then, the following holds: $S(v, v_0) = \{ \emptyset \}$, $S(v, v_1) = \{ \emptyset, \{ u_A \}, \{ u_B \} \}$, $S(v, v_2) = \{ \emptyset, \{ u_C \} \}$, $S(v, v_3) = \{ \emptyset, \{ u_D \}, \{ u_E \} \}$, $S(v, v_4) = \{ \emptyset, \{ u_D \}, \{ u_E \}, \{ v_F \}, \{ u_D, u_E \}, \{ u_D, u_F \}, \{ u_E, u_F \} \}$.

\textbf{Proposition 4.} $S(v) = \cup_{v_i \in \text{Des}(v)} S(v, v_i)$.

\textbf{Proof.} Let $U \in S(v)$ and $d_U = |U|$. Let $\phi$ be an injection from $U$ to $\text{Des}(v)$ giving an inclusion mapping for $P(U) \subset T(v)$. Let $\{ v_1', \ldots, v_{d_U}' \} = \{ \phi(u_j) \mid u_j \in U \}$, where $v_1' \triangleleft v_2' \triangleleft \cdots \triangleleft v_{d_U}'$. Then, $v_i' \in LF(v, v_{i+1}')$ and $v_i' \in LF(v, v_{i+1}')$ hold for all $i = 1, \ldots, d_U - 1$. Furthermore, $P(u_i) \sim T(v_i')$ holds for all $i = 1, \ldots, d_U - 1$. Therefore, $U \in S(v, v_{d_U}')$.

It is straightforward to see that $S(v, v_i)$ does not contain any element not in $S(v)$. ▶
Lemma 5. \textbf{AlgInc1} correctly computes $S(v,v_i)$ for all $v_j \in Des(v)$ in $O(d2^d n^2)$ time.

\textbf{Proof.} Since it is straightforward to prove the correctness, we analyze the time complexity. The sizes of $S(v)$, $S(v,v_i)$s, and $S_0(v_i)$s are $O(d2^d)$, and computation of each of such sets can be done in $O(d2^d n)$ time. Since the number of $S(v,v_i)$s and $S_0(v_i)$s (per $v$) are $O(n)$, the total computation time is $O(d2^d n^2)$.

If there exist $u_i,u_j \in Chd(u)$ such that $P(u_i) \sim P(u_j)$, we treat each element in $S(v)$, $S(v,v_i)$s, and $S_0(v_i)$s as a multiset where any $u_i$ and $u_j$ such that $P(u_i) \sim P(u_j)$ are identified and the multiplicity of $u_i$ is bounded by the number of $P(u_j)$s isomorphic to $P(u_i)$. Then, since $|Chd(u)| \leq d$ for all $u$ in $P$, the size of each multiset is at most $d$ and the number of different multisets is not greater than $2^d$. Therefore, the same time complexity result
holds. This discussion can also be applied to the following sections. Note that by treating these \( u_i \) and \( u_j \) separately, we need not change the algorithm. However, use of multi-sets plays an important role in Section 7.

**Theorem 7.** AlgInc1 does a lot of redundant computations. In order to compute \( S_0(v_i) \), we do not need to consider all \( v_s \) that are left of \( v_i \). Instead, we construct a tree \( T'(v) \) from a given \( T(v) \) by the following rule: for each pair of consecutive siblings \( (v_i, v_j) \) in \( T(v) \), add a new sibling (leaf) \( v_{(i,j)} \) between \( v_i \) and \( v_j \). Newly added nodes are called virtual nodes. We construct a DAG \( G'(V', E') \) on \( V' = V(T'(v)) \) by: \( (v_i, v_j) \in E' \) iff one of the following holds
- \( v_j \) is a virtual node, and \( v_i \) is in the rightmost path of \( T'(v_{j1}) \), where \( v_j = v_{(j1,j2)} \).
- \( v_i \) is a virtual node, and \( v_j \) is in the leftmost path of \( T'(v_{j2}) \), where \( v_i = v_{(i1,i2)} \).

Then, we can use the same technique as AlgInc1, except that \( G(V, E) \) is replaced by \( G'(V', E') \). We denote the resulting algorithm by AlgInc2.

**Lemma 6.** AlgInc2 correctly computes \( S(v, v_j) \) for all \( v_j \in Des(v) \) in \( O(d2^d n) \) time.

Since checking the minimality can be done in \( O(m) \) time per \((u, v)\), it is seen from Proposition 3 that the total time complexity is \( O(d2^d m^2) \). Since the size of each \( S(v, v_j) \) is \( O(d2^d) \) and we need to maintain information about \( P(u) \prec T(v) \) and \( P(u) \subset T(v) \) for all \((u, v)\), the total space is \( O(d2^d n + mn) \).

**Theorem 7.** Unordered tree inclusion can be solved in \( O(d2^d m^2 n) \) time using \( O(d2^d n + mn) \) space.

If we analyze the time complexity carefully, we can see that it is \( O(d2^d h(T) mn) \) because each \( v_i \) is involved in computation of \( P(u) \prec T(v) \) only for \( v \in Anc(v_i) \). This result is better than that of [14] if \( d \) is not small (precisely, \( d > c \log(h(T)) \) for some constant \( c \)).

## 4 NP-hardness of unordered tree inclusion for pattern trees with unique leaf labels

For any node-labeled tree \( T \), let \( L(T) \) be the set of all leaf labels in \( T \). For any \( c \in L(T) \), let \( OCC(T, c) \) be the number of times that \( c \) occurs in \( T \), and define \( OCC(T) = \max_{c \in L(T)} OCC(T, c) \).

The decision version of the tree inclusion problem is to determine whether \( T \) can be obtained from \( P \) by applying node insertion operations. Kihheläinen and Mannila [14] proved that the decision version of unordered tree inclusion is NP-complete by reducing from satisfiability. In their reduction, the clauses in a given instance of satisfiability are represented by node labels in the constructed trees; in particular, for every clause \( C \), each literal in \( C \) introduces one node in \( T \) whose node label represents \( C \). By using 3-SAT instead of satisfiability in their reduction, we immediately have:

**Corollary 8.** The decision version of the unordered tree inclusion problem is NP-complete even if restricted to instances where \( h(T) = 2, h(P) = 1, OCC(T) = 3, \) and \( OCC(P) = 1 \).

In Kihheläinen and Mannila’s reduction, the labels assigned to the internal nodes of \( T \) are significant. Here, we consider the computational complexity of the special case of the problem where all internal nodes in \( P \) and \( T \) have the same label, or equivalently, where only the leaves are labeled. Then, we have the following.

**Theorem 9.** The decision version of the unordered tree inclusion problem is NP-complete even if restricted to instances where \( h(T) = 2, h(P) = 2, OCC(T) = 3, OCC(P) = 1, \) and all internal nodes have the same label.
Figure 3 For these trees, $\text{Occ}(u_1, M) = \text{Occ}(u_2, M) = 3$, $\text{Occ}(u_3, M) = \text{Occ}(u_4, M) = \text{Occ}(u_5, M) = 2$, $d_2 = 3$, $d_3 = 2$, and $\text{OCC}(P, T) = 3$.

5 A polynomial-time algorithm for case of $\text{OCC}(P, T) = 2$

In this and the following sections, for the simplicity, we consider the decision version of unordered tree inclusion. However, by repeatedly applying each procedure $O(n)$ times, we can solve the locating problem version and thus the theorems hold as they are.

In this section, we require that each leaf of $P$ has a unique label and that it appears at no more than $k$ leaves in $T$. We denote this number $k$ by $\text{OCC}(P, T)$ (see Figure 3). Note that the case of $\text{OCC}(P) = 1$ and $\text{OCC}(T) = k$ is included in the case of $\text{OCC}(P, T) = k$. From the unique leaf label assumption, we have the following observation.

- Proposition 10. Suppose that $P(u)$ has a leaf labeled with $b$. If $P(u) \subset T(v)$, then $v$ is an ancestor of a leaf (or leaf itself) with label $b$.

We say that $v_j$ is a minimal node for $u_i$ if $P(u_i) \prec T(v_j)$ holds. It follows from this proposition that the number of minimal nodes is at most $k$ for each $u_i$ if $\text{OCC}(P, T) = k$.

When $k = 2$, we can have a chain of choices of the subtrees of $P$ in $T$. This suggests that 2-SAT is useful. Indeed, by using a polynomial-time reduction to 2-SAT, we have:

- Theorem 11. Unordered tree inclusion can be solved in polynomial time if $\text{OCC}(P, T) = 2$.

6 An $O^*(1.8^d)$-time algorithm for case of $\text{OCC}(P, T) = 3$

In this section, we present an $O^*(1.8^d)$-time algorithm for the case of $\text{OCC}(P, T) = 3$, where $d$ is the maximum degree of $P$, $m = |V(P)|$, and $n = |V(T)|$. Note that this case remains NP-hard from Theorem 9.

The basic strategy is use of dynamic programming: decide whether $P(u) \subset T(v)$ in a bottom-up way. Suppose that $u$ has a set of children $U = \{u_1, \ldots, u_d\}$. Since we use dynamic programming, we can assume that $P(u_i) \prec T(v_j)$ is known for all $u_i$ and for all $v_j \in V(T(v)) - \{v\}$. We define $M(u, v)$ by $M(u, v) = \\{(u_i, v_j) | P(u_i) \prec T(v_j) \land v_j \in V(T(v))\}$.

The crucial task of the dynamic programming procedure is to find an injective mapping $\psi$ from $\{u_1, \ldots, u_d\}$ to $V(T(v)) - \{v\}$ such that $P(u_i) \prec T(\psi(u_i))$ holds for all $u_i$ ($i = 1, \ldots, d$) and there is no ancestor/descendant relationship between any $\psi(u_i)$ and $\psi(u_j)$ ($u_i \neq u_j$). If this task can be performed in $O(f(d, m, n))$ time, from Proposition 3, the total complexity will be $O^*(f(d, m, n))$. We assume w.l.o.g. that $\psi$ is given as a set of mapping pairs. For each $v_j \in V(T(v))$ and each $M \subseteq M(u, v)$, we define $\text{AncDes}(v_j, T, M)$ by

\[ \text{AncDes}(v_j, T, M) = \{ (u_k, v_h) | (u_k, v_h) \in M \land v_h \in (\{v_j\} \cup \text{Anc}(v_j, T) \cup \text{Des}(v_j, T)) \}, \]

where $\text{Anc}(v_j, T)$ (resp., $\text{Des}(v_j, T)$) denotes the set of ancestors (resp., descendants) of $v_j$ in $T$ where $v_j \notin \text{Anc}(v_j, T)$ (resp., $v_j \notin \text{Des}(v_j, T)$).
Here, we define $\text{Occ}(u_i, M)$ by $\text{Occ}(u_i, M) = |\{j \mid (u_i, v_j) \in M\}|$, where $M = M(u, v)$. Let $d_3$ (resp., $d_2$) be the number of $u_i$s such that $\text{Occ}(u_i, M) = 3$ (resp., $\text{Occ}(u_i, M) = 2$) (see also Figure 3). We assume w.l.o.g. that $d_2 + d_3 = d$ because $\text{Occ}(u_i, M) = 1$ means that $\psi(u_i)$ is uniquely determined and thus we can ignore $u_i$s with $\text{Occ}(u_i, M) = 1$. From Theorem 11, we can see the following if there are no two pairs $(u_{i_1}, v_{j_1}), (u_{i_2}, v_{j_2}) \in M$ such that $\text{Occ}(u_{i_1}, M) = 3$, $\text{Occ}(u_{i_2}, M) = 3$, and $(u_{i_2}, v_{j_2}) \in \text{AncDes}(v_{j_1}, T(v), M)$.

- The problem can be solved in $O^*(2^{d_2})$ time:

  For each $u_i$ such that $\text{Occ}(u_i, M) = 3$ (i.e., $(u_i, v_{j_1}), (u_i, v_{j_2}), (u_i, v_{j_3}) \in M$), we choose $\psi(u_i)$ (i.e., $(u_i, v_{j_1}) \in \psi$ or not). Thus, there exist $2^{d_2}$ possibilities. After all the choices, there is no $u_i$ such that $\text{Occ}(u_i, M) = 3$ and Theorem 11 can be applied.

- The problem can also be solved in $O^*(2^{d_2})$ time:

  For each $u_i$ with $\text{Occ}(u_i, M) = 2$ (i.e., $(u_i, v_{j_1}), (u_i, v_{j_2}) \in M$), we must choose $\psi(u_i) = v_{j_1}$ or $\psi(u_i) = v_{j_2}$. Thus, there are $2^{d_2}$ possibilities. After all choices, each $(u_i, v_j) \in M$ with $\text{Occ}(u_i, M) = 2$ is removed, and thus there is no pairs $(u_{i_1}, v_{j_1}), (u_{i_2}, v_{j_2}) \in M$ such that $(u_{i_2}, v_{j_2}) \in \text{AncDes}(v_{j_1}, T(v), M)$ from the ‘if’ condition. The problem is reduced to bipartite matching, which can be solved in polynomial time.

It means the problem can be solved in $O^*(\min(2^{d_2}, 2^{d_2}))$ time. We denote the condition (i.e., ‘if’ part of the above) and this algorithm by $(\# \#)$ and ALG-##, respectively. Therefore, the crucial point is how to (recursively) remove pairs such that $\text{Occ}(u_{i_1}, M) = 3$, $\text{Occ}(u_{i_2}, M) = 3$, and $(u_{i_2}, v_{j_2}) \in \text{AncDes}(v_{j_1}, T(v), M)$.

For a mapping $\psi$, we let $\psi \cup \text{NULL} = \text{NULL}$, where NULL means that there is no valid mapping. The following is a pseudocode of the algorithm for finding a mapping $\psi$, where it is invoked as $\text{FindMapping}(u_1, \ldots, u_d, M)$ with $M = M(u, v)$.

Procedure $\text{FindMapping}(U, M)$

if condition (##) is satisfied then
  return mapping by ALG-(##);
else
  Choose arbitrary $(u_{i_1}, v_{j_1}), (u_{i_2}, v_{j_2}) \in M$ such that $\text{Occ}(u_{i_1}, M) = 3$, $\text{Occ}(u_{i_2}, M) = 3$, and $(u_{i_2}, v_{j_2}) \in \text{AncDes}(v_{j_1}, T(v), M)$;
  $M' \leftarrow M - \{(u_{i_1}, v_{j_1})\}$;
  $\psi \leftarrow \text{FindMapping}(U, M')$;
  if $\psi \neq \text{NULL}$ return $\psi$;
  $M' \leftarrow M - \text{AncDes}(v_{j_1}, T(v), M)$;
  return $(u_{i_1}, v_{j_1}) \cup \text{FindMapping}(U - \{u_{i_1}\}, M')$.

$\triangleright$ Theorem 12. Unordered tree inclusion can be solved in $O^*(1.8^d)$ time if $\text{Occ}(P, T) = 3$.

7 A randomized algorithm for case of $h(P) = 1$ and $h(T) = 2$

In this section, we consider the case of $h(P) = 1$ and $h(T) = 2$, which is denoted by IncH2 and remains NP-hard from Corollary 8. We assume w.l.o.g. that the roots of $P$ and $T$ have the same unique label and thus they must match in any inclusion mapping.

Let $U = \{u_1, \ldots, u_d\}$ be the set of children of $r(P)$. Let $v_1, \ldots, v_d$ be the children of $r(T)$, and let $v_{i_1}, \ldots, v_{i_{n_i}}$ be the children of each $v_i$.

First, we assume that $\ell(u_i) \neq \ell(u_j)$ holds for all $i \neq j$, where $\ell(v)$ denotes the label of $v$. This special case is denoted by IncH2U. Recall that IncH2U remains NP-hard.
IncH2U can be solved by a reduction to CNF SAT, which is different from the one in Section 5 and is considered as a reverse reduction of the one used for proving NP-hardness of unordered tree inclusion [14]. For each \( u_i \), we define \( X_i^{POS} \) and \( X_i^{NEG} \) by

\[
X_i^{POS} = \{ x_j | \ell(u_i) = \ell(v_j) \}, \quad X_i^{NEG} = \{ x_j | (\exists v_{j,k} \in Chd(v_j))(\ell(u_i) = \ell(v_{j,k})) \}.
\]

For each \( u_i \), we construct a clause \( C_i \) by

\[
C_i = \left( \bigvee_{x_j \in X_i^{POS}} x_j \right) \lor \left( \bigvee_{x_j \in X_i^{NEG}} \overline{x_j} \right).
\]

Then, the resulting SAT instance is \( \{ C_1, \ldots, C_d \} \). Intuitively, \( x_j = 1 \) corresponds to a case that \( u_i \) is mapped to \( v_j \), where \( \ell(u_i) = \ell(v_j) \). Of course, multiple \( v_j \)'s may correspond to \( u_i \). However, it is enough to consider an arbitrary one.

Then, it is straightforward to see that \( P \) is included in \( T \) iff \( \{ C_1, \ldots, C_d \} \) is satisfiable. Using Yamamoto’s algorithm for SAT with \( d \) clauses [22], we have:

**Proposition 13.** IncH2U can be solved in \( O^*(1.234^d) \) time.

Next, we consider IncH2. We combine two algorithms: (A1) random sampling-based algorithm, and (A2) modified version of the \( O(d2^dmn^2) \) time algorithm in Section 3.

For (A1), we employ a technique used in color-coding [6]. Let \( d_0 \) be the number of \( u_i \)'s having unique labels. Let \( d_1 \leq d_2 \leq \cdots \leq d_h \) be the multiplicities of other labels in \( U \). Note that \( d_0 + d_1 + \cdots + d_h = d \) holds. Let \( d - d_0 = \alpha d \).

For each label \( a_i \) with \( d_i > 1 \) (i.e., \( i > 0 \)), we change the labels of nodes with label \( a_i \) in \( P \) to \( a_1, a_2, \ldots, a_h \) in an arbitrary way. For each node \( v \) in \( T \) having label \( a_i \), we assign \( a_j^t \) (\( j = 1, \ldots, d_i \)) to \( v \) uniformly at random, and then apply the SAT-based algorithm for IncH2U. Let \( M \) be the set of pairs for an inclusion mapping from \( P \) to \( T \). If all nodes of \( T \) appearing in \( M \) have different labels, a valid inclusion mapping can be obtained. This success probability is given by

\[
\frac{d_1! \cdots d_2! \cdots d_h!}{d_1^{d_1} \cdots d_2^{d_2} \cdots d_h^{d_h}} \geq \frac{(d_1 + d_2)!}{(d_1 + d_2)^{d_1 + d_2}}.
\]

Note that this inequality is proved by repeatedly applying \( \frac{d_1!}{d_1^{d_1}} \geq \frac{d_1 + d_2}{d_1} \) \( d_2 \)-times, which is seen from

\[
\frac{(d_1 + d_2)^{d_1 + d_2}}{d_1^{d_1} d_2^{d_2}} \geq \frac{(d_1 + d_2)!}{(d_1 + d_2)^{d_1 + d_2}}.
\]

If we repeat the random sampling procedure \( e^{\alpha d} \) times, the failure probability is at most \( (1 - e^{-\alpha d})^{e^{\alpha d}} \leq e^{-1} < \frac{1}{2} \).

For (A2), we modify the \( O(d2^dmn^2) \) time algorithm as follows. Recall that if there exist labels with multiplicity more than one, \( S(v, v_i) \) is a multi-set. In order to represent a multi-set, we memorize the multiplicity of each label. Then, the number of distinct multi-sets is given by

\[
N(d_0, \ldots, d_h) = 2^{d_0} \cdot \prod_{l=1}^{h} (d_l + 1).
\]

Since \( d_i + 1 \leq 3^{(d_i/2)} \) holds for any \( d_i \geq 2 \), this number is bounded as

\[
N(d_0, \ldots, d_h) \leq 2^{d_0} \cdot 3^{(d - d_0)/2}.
\]

Then, the time complexity of (A2) is \( O^*(2^{(1-\alpha)d} \cdot 3^{(\alpha/2)d}) \).
Since we can use the minimum of the time complexities of (A1) and (A2), the resulting time complexity is given by
\[
\max_{\alpha} \min\left( O^*(1.234^d \cdot e^{\alpha d}), O^*(2(1-\alpha)^{d} \cdot 3^{(\alpha/2)d}) \right).
\]

By numerical calculation, this is \( O^*(1.883^d) \).

\[\blacktriangleright\textbf{Theorem 14.} \textit{IncH2} can be solved in randomized \( O^*(1.883^d) \) time with probability at least \( 1 - \frac{1}{n^k} \), where \( k \) is any positive constant. \]

It seems that the above algorithm can be de-randomized by using the \( k \)-perfect hash family as in [6]. However, since the construction of a \( k \)-perfect hash family has a high complexity, the resulting algorithm might have a time complexity much worse than \( O^*(2^d) \).

\[\blacktriangleright\textbf{Concluding remarks} \]

We have improved the exponential factor of Kilpeläinen and Mannila’s [14] well-known algorithm from 1995 for unordered tree inclusion from \( 2^{2d} \) to \( 2^d \). Observe that the \( 2^d \) factor may not be optimal. Indeed, we have presented a randomized \( O^*(1.883^d) \)-time algorithm for the case of \( h(P) = 1 \) and \( h(T) = 2 \). However, we could not obtain an \( O^*((2 - \epsilon)^d) \)-time algorithm for any constant \( \epsilon > 0 \) even for the case of \( h(P) = h(T) = 2 \). Development of an \( O^*((2 - \epsilon)^d) \)-time algorithm for unordered tree inclusion, or showing an \( \Omega(2^d) \) lower bound using recent techniques for proving lower bounds on various matching problems [1, 2, 9], is left as an open problem.

\[\blacktriangleright\textbf{References} \]


Beyond-Planarity: Turán-Type Results for Non-Planar Bipartite Graphs

Patrizio Angelini
Wilhelm-Schickhard-Institut für Informatik, Universität Tübingen, Germany

Michael A. Bekos
Wilhelm-Schickhard-Institut für Informatik, Universität Tübingen, Germany

Michael Kaufmann
Wilhelm-Schickhard-Institut für Informatik, Universität Tübingen, Germany

Maximilian Pfister
Wilhelm-Schickhard-Institut für Informatik, Universität Tübingen, Germany

Torsten Ueckerdt
Fakultät für Informatik, KIT, Karlsruhe, Germany

Abstract

Beyond-planarity focuses on the study of geometric and topological graphs that are in some sense nearly planar. Here, planarity is relaxed by allowing edge crossings, but only with respect to some local forbidden crossing configurations. Early research dates back to the 1960s (e.g., Avital and Hanani 1966) for extremal problems on geometric graphs, but is also related to graph drawing problems where visual clutter due to edge crossings should be minimized (e.g., Huang et al. 2018).

Most of the literature focuses on Turán-type problems, which ask for the maximum number of edges a beyond-planar graph can have. Here, we study this problem for bipartite topological graphs, considering several types of beyond-planar graphs, i.e. 1-planar, 2-planar, fan-planar, and RAC graphs. We prove bounds on the number of edges that are tight up to additive constants; some of them are surprising and not along the lines of the known results for non-bipartite graphs. Our findings lead to an improvement of the leading constant of the well-known Crossing Lemma for bipartite graphs, as well as to a number of interesting questions on topological graphs.

2012 ACM Subject Classification Theory of computation → Computational geometry, Mathematics of computing → Graph theory

Keywords and phrases Bipartite topological graphs, beyond planarity, density, Crossing Lemma

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.28

Related Version A full version of the paper is available at [9], https://arxiv.org/abs/1712.09855.

Funding This project was supported by DFG grant KA812/18-1.

1 Introduction

Planarity has been a central concept in the areas of graph algorithms, computational geometry, and graph theory since the beginning of the previous century. While planar graphs were originally defined in terms of their geometric representation, they exhibit a number of combinatorial properties that only depend on their abstract representations. To mention some of the most important landmarks, we refer to the characterization of planar graphs in...
Beyond-Planarity: Turán-Type Results for Non-Planar Bipartite Graphs

(a) 1-planar  (b) 3-quasiplanar  (c) fan-planar  (d) RAC

Figure 1 Different forbidden crossing configurations.

terms of forbidden minors, to the existence of linear-time algorithms to test planarity, to the Four-Color theorem, and to the Euler’s polyhedron formula, which implies that n-vertex planar graphs have at most $3n - 6$ edges.

For the applicative purpose of visualizing real-world networks, however, the concept of planarity turns out to be overly restrictive. Graphs representing such networks are too dense to be planar, even though one can often confine non-planarity in some local structures. Also, cognitive experiments [28] show that this does not affect the readability of the drawing too much, if these local structures satisfy specific properties. In other words, these experiments indicate that even non-planar drawings may be effective for human understanding, as long as the crossing configurations satisfy certain properties. Different requirements on the crossing configurations naturally give rise to different classes of beyond-planar graphs. Beyond-planarity is then defined as a generalization of planarity, which encompasses all these classes. Early works date back to 60’s [12] in the field of extremal graph theory, and continued over the years [3, 7, 33]; also due to the aforementioned experiments, a strong attention on the topic was recently raised (e.g., [21]), which led to many results, described below.

Some of the most studied beyond-planar graphs include:

(i) **k-planar graphs**, in which each edge is crossed at most $k$ times [2, 32, 33], see Fig. 1a;

(ii) **k-quasiplanar graphs**, which disallow sets of $k$ pairwise crossing edges [1, 3, 24], see Fig. 1b;

(iii) **fan-planar** graphs, in which no edge is crossed by two independent edges or by two adjacent edges from different directions [14, 30], see Fig. 1c;

(iv) **RAC graphs**, in which crossings happen at right angles [20, 22]; see Fig. 1d.

Two notable sub-families of 1-planar graphs are the **IC-planar graphs** [38], where crossings are independent (i.e., no two crossed edges share an endpoint), and the **NIC-planar graphs** [37], where crossings are nearly independent (i.e., no two pairs of crossed edges share two endpoints). For a survey providing an overview on beyond-planarity see [21].

From the combinatorial point of view, the main extremal graph theory question, also called Turán-type [15], concerns the maximum number of edges for graphs in a certain class. Tight density bounds are known for several classes [20, 30, 33, 37, 38]; a main open question is to determine the density of $k$-quasiplanar graphs, which is conjectured to be linear in $n$ for any fixed $k$ [1, 3, 7, 24]. The new bounds for 1-, 2-, 3- and 4-planar graphs have led to progressive improvements on the leading constant of the lower bound on the number of crossings of a graph, provided by the well-known Crossing Lemma, from $\frac{1}{300} = 0.01$ [5, 31] to $\frac{1}{447} \approx 0.0023$ [4], to $\frac{1}{327} \approx 0.00296$ [33], to $\frac{1}{31} \approx 0.0322$ [32], to $\frac{1}{30} \approx 0.0345$ [2]. Related combinatorial problems concern inclusion relationships between classes [8, 14, 17, 22, 25].

From the complexity side, in contrast to efficient planarity testing algorithms [27], recognizing a beyond-planar graph has often been proven to be NP-hard [10, 14]. Polynomial-time testing algorithms can be found when posing additional restrictions on the produced drawings, namely, that the vertices are required to lie either on two parallel lines (see, e.g., [14, 19]) or on the outer face of the drawing (see, e.g., [11, 26]).
Table 1 Summary of our results (from sparse to dense); the bound with asterisk (*) is not tight.

<table>
<thead>
<tr>
<th>Graph class</th>
<th>General Bound (tight)</th>
<th>Ref.</th>
<th>Bipartite Lower bound</th>
<th>Ref.</th>
<th>Upper bound</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC-planar:</td>
<td>3.5n − 7</td>
<td>[38]</td>
<td>2.25n − 4</td>
<td>Thm.1</td>
<td>2.25n − 4</td>
<td>Thm.2</td>
</tr>
<tr>
<td>NIC-planar:</td>
<td>3.6n − 7.2</td>
<td>[37]</td>
<td>2.5n − 5</td>
<td>Thm.1</td>
<td>2.5n − 5</td>
<td>Thm.3</td>
</tr>
<tr>
<td>1-planar:</td>
<td>4n − 8</td>
<td>[34]</td>
<td>3n − 8</td>
<td>[18]</td>
<td>3n − 8</td>
<td>[18]</td>
</tr>
<tr>
<td>RAC:</td>
<td>4n − 10</td>
<td>[20]</td>
<td>3n − 9</td>
<td>Thm.4</td>
<td>3n − 7</td>
<td>Thm.5</td>
</tr>
<tr>
<td>2-planar:</td>
<td>5n − 10</td>
<td>[33]</td>
<td>3.5n − 12</td>
<td>Thm.13</td>
<td>3.5n − 7</td>
<td>Thm.15</td>
</tr>
<tr>
<td>fan-planar:</td>
<td>5n − 10</td>
<td>[30]</td>
<td>4n − 16</td>
<td>Thm.6</td>
<td>4n − 12</td>
<td>Thm.11</td>
</tr>
<tr>
<td>3-planar:</td>
<td>5.5n − 11</td>
<td>[32]</td>
<td>4n − O(1)</td>
<td>Sec.6</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>k-planar:</td>
<td>3.81√kn *</td>
<td>[2]</td>
<td>—</td>
<td>—</td>
<td>3.005√kn</td>
<td>Thm.17</td>
</tr>
</tbody>
</table>

Another natural restriction, yet rarely explored in the literature, is to pose additional structural constraints on the graphs themselves, rather than on their drawings. For 3-connected 1-plane graphs, Alam et al. [6] presented a polynomial-time algorithm to construct planar straight-line drawings. Further, Brandenburg [16] gave an efficient algorithm to recognize optimal 1-planar graphs, i.e., those with the maximum number of edges.

For the important class of bipartite graphs, very few results have been discovered so far. From the density point of view, the only result we are aware of is a tight bound of $3n − 8$ edges for bipartite 1-planar graphs [18, 29]. Didimo et al. [19] characterize the complete bipartite graphs that admit RAC drawings, but their result does not extend to non-complete graphs.

Our contribution. Along this direction, we study several classes of beyond-planar bipartite topological or geometric graphs, focusing on Turán-type problems. Table 1 shows our findings. The new bound on the edge density of bipartite 2-planar graphs leads to an improvement of the leading constant of the Crossing Lemma for bipartite graphs from $\frac{1}{29} \approx 0.0345$, which holds for general graphs [2], to $\frac{1}{18.1} \approx 0.0554$ (see Theorem 16). To the best of our knowledge, this is the first non-trivial adjustment of the Crossing Lemma that is specific for bipartite graphs, besides the Zarankiewicz conjecture [36], which however only concerns complete bipartite graphs. Our results also unveil an interesting tendency in the density of $k$-planar bipartite graphs with respect to the one of general $k$-planar graphs. At first sight, the differences seem to be around $n$, as it is in the planar and in the 1-planar cases (i.e., $n − 2$). This turns out to be true also for RAC and fan-planar graphs. However, for IC- and NIC-planar graphs, and in particular for 2-planar graphs, the differences are larger.

Another notable observation from our results is that, in the bipartite setting, fan-planar graphs can be denser than 2-planar graphs, while in the non-bipartite case these two classes have the same maximum density, even though none of them is contained in the other [14]. In Section 6 we discuss a number of open problems that are raised by our work.

Methodology. We focus on five classes of bipartite beyond-planar graphs; see Sections 2–5. To estimate the maximum edge density of each class we employ different counting techniques.

- For the class of bipartite IC-planar graphs, we apply a direct counting argument based on the number of crossings that are possible due to the restrictions posed by IC-planarity.
- Our approach is different for NIC-planarity. We show that a bipartite NIC-planar graph of maximum density contains a set of uncrossed edges forming a plane subgraph whose faces have length 6, and that each such face contains exactly one crossing pair of edges.
Beyond-Planarity: Turán-Type Results for Non-Planar Bipartite Graphs

To estimate the maximum number of edges of a bipartite RAC graph, we adjust a technique by Didimo et al. [20], who proved the corresponding bound for general RAC graphs.

For fan-planarity, our technique is more involved. After examining structural properties of maximal bipartite fan-planar graphs, we show how to augment them so that they contain a planar quadrangulation as a subgraph. Then, we develop a charging scheme which charges edges involved in fan crossings to the corresponding vertices, to prove that there are at least as many edges in the quadrangulation as in the rest of the graph.

For 2-planarity, we again show that maximal bipartite 2-planar graphs have a planar quadrangulation as a subgraph. We then use a counting scheme based on an auxiliary directed plane graph, defined by orienting the dual of the quadrangulation, describing dependencies of adjacent quadrangular faces posed by the edges not belonging to it.

Preliminaries. We consider connected topological graphs, i.e., drawn in the plane with vertices represented by points in \( \mathbb{R}^2 \) and edges by Jordan arcs connecting their endvertices, so that:

(i) no edge passes through a vertex different from its endpoints,
(ii) no two adjacent edges cross,
(iii) no edge crosses itself,
(iv) no two edges meet tangentially, and
(v) no two edges cross more than once.

A graph has no self-loops or multiedges. Otherwise, it is a topological multigraph, for which we assume that the two regions defined by self-loops or multiedges contain at least one vertex in their interiors, i.e., all edges are non-homotopic.

We refer to a beyond-planar graph \( G \) with \( n \) vertices and maximum possible number of edges as optimal. Consider all the plane spanning subgraphs of \( G \) (i.e., in their drawings inherited from \( G \) there exists no two crossing edges). Among those, we select one with the largest number of edges, which we denote by \( G_p \) and call it the planar structure of \( G \).

\[ f = \langle u_0, u_1, \ldots, u_{k-1} \rangle \]

is a face of \( G_p \). We say that \( f \) is simple if \( u_i \neq u_j \) for each \( i \neq j \); face \( f \) is connected if edge \( (u_i, u_{i+1}) \) exists for each \( i = 0, \ldots, k - 1 \) (indices modulo \( k \)).

2 Bipartite IC- and NIC-Planar Graphs

In this section, we give tight bounds on the density of bipartite IC- and NIC-planar graphs. For the proofs of the lower bounds, we refer to Fig. 2. Full proofs can be found in [9].

- Theorem 1. For infinitely many values of \( n \), there exists a bipartite \( n \)-vertex IC-planar graph with \( 2.25n - 4 \) edges, and a bipartite \( n \)-vertex NIC-planar graph with \( 2.5n - 5 \) edges.

- Theorem 2. A bipartite \( n \)-vertex IC-planar graph has at most \( 2.25n - 4 \) edges.

Proof. Our proof is an adjustment of the one for general IC-planar graphs [38]. Let \( G \) be a bipartite \( n \)-vertex optimal IC-planar graph. Let \( cr(G) \) be the number of crossings of \( G \). Since every vertex of \( G \) is incident to at most one crossing, \( cr(G) \leq \frac{n}{4} \). By removing one edge

![Figure 2 Bipartite \( n \)-vertex IC- and NIC-planar graphs with (a) 2.25\( n \) - 4 and (b) 2.5\( n \) - 5 edges.](image-url)
from every pair of crossing edges of $G$, we obtain a plane bipartite graph, which has at most $2n - 4$ edges. Hence, the number of edges of $G$ is at most $2n - 4 + cr(G) = 2.25n - 4$. ▶

**Theorem 3.** A bipartite $n$-vertex NIC-planar graph has at most $2.5n - 5$ edges.

**Proof.** Among all bipartite optimal NIC-planar graphs with $n$ vertices, let $G$ be the one with the maximum number of uncrossed edges, i.e., $G$ is such that the plane (bipartite) subgraph $H$ obtained by removing every crossed edge in $G$ has maximum density. It is not difficult to show that each face of $H$ containing two crossing edges in $G$ is connected and has length 6 (for details see [9]). Thus, every face of $H$ has length either 6, if it contains two edges crossing in $G$, or 4 otherwise (due to bipartiteness and maximality).

Let $\nu$ and $\mu$ be the number of vertices and edges of $H$, respectively. Clearly, $n = \nu$. Let also $\phi_4$ and $\phi_6$ be the number of faces of length 4 and 6 in $H$, respectively. We have that $2\phi_4 + 3\phi_6 = \mu$. By Euler’s formula, we also have that $\mu + 2 = \nu + \phi_4 + \phi_6$. Combining these two equations, we obtain: $\phi_4 + 2\phi_6 = \nu - 2$. So, in total the number of edges of $G$ is $\mu + 2\phi_6 = 2\phi_4 + 5\phi_6 = 2n - 4 + \phi_6$. By Euler’s formula, the number of faces of length 6 of a planar graph is at most $(n - 2)/2$, which implies that $G$ has at most $2.5n - 5$ edges. ▶

### 3 Bipartite RAC Graphs

We continue our study on bipartite beyond-planarity with the class of geometric RAC graphs. We prove an upper bound on their density that is optimal up to a constant of 2.

**Theorem 4.** For infinitely many values of $n$, there exists a bipartite $n$-vertex RAC graph with $3n - 9$ edges.

**Proof.** For any $k > 1$, we recursively define a graph $G_k$ by attaching six vertices and 18 edges to $G_{k-1}$; see the left part of Fig. 3. The base graph $G_1$ is a hexagon containing two crossing edges. So, $G_k$ has $6k$ vertices and $18k - 10$ edges. The right part of Fig. 3 shows that $G_k$ is RAC: if $G_{k-1}$ is drawn so that its outerface is a parallelogram, then it can be augmented to a RAC drawing of $G_k$ whose outerface is a parallelogram with sides parallel to the ones of $G_{k-1}$. The bound follows by adding an edge in the outerface of $G_k$ by slightly “adjusting” its drawing; see [9].

**Theorem 5.** A bipartite $n$-vertex RAC graph has at most $3n - 7$ edges.

**Proof.** Let $G$ be a (possibly non-bipartite) RAC graph with $n$ vertices. Since $G$ does not contain three mutually crossing edges, as in [20] we can color its edges with three colors (r, b, g) so that the crossing-free edges are the r-edges, while b-edges cross only g-edges, and vice-versa. Thus, the subgraphs $G_{rb}$, consisting of only r- and b-edges, and $G_{rg}$, consisting of only r- and g-edges, are both planar. Didimo et al. [20, Lemma 4] showed that each face of $G_{rb}$ has at least two r-edges, by observing that if this property did not hold, then the drawing could be augmented by adding r-edges. Thus, the number $m_b$ of b-edges is at most
Beyond-Planarity: Turán-Type Results for Non-Planar Bipartite Graphs

Figure 4 (a) $K_{4,n-4}$ with four additional multiedges (thick), (b) $K_{2,n-2}$ with $2n - 8$ additional multiedges (thick), and (c) $K_{5,5} - e$ with four additional multiedges (thick).

$n - 1 - \lfloor \lambda/2 \rfloor$, where $\lambda \geq 3$ is the number of edges in the outer face of $G$. Suppose now that $G$ is additionally bipartite. We still have $m_b \leq n - 1 - \lfloor \lambda/2 \rfloor$, but in this case $\lambda \geq 4$ holds (by bipartiteness). Hence, $m_b \leq n - 3$. Since $G_{e_2}$ is bipartite and planar, it has at most $2n - 4$ edges (i.e., $m_r + m_q \leq 2n - 4$). Hence, $G$ has at most $3n - 7$ edges. ▶

4 Bipartite Fan-Planar Graphs

We continue our study with the class of fan-planar graphs. We begin as usual with the lower bound (Theorem 6), which we suspect to be best-possible both for graphs and multigraphs. For fan-planar bipartite graphs, we prove an almost tight upper bound (Theorem 11).

Theorem 6. For infinitely many values of $n$, there exists a bipartite $n$-vertex fan-planar graph with $4n - 14$ edges, and

(i) graph with $4n - 16$ edges, and

(ii) multigraph with $4n - 12$ edges.

Proof sketch. Figs. 4a, 4b, and 4c show constructions that yield bipartite $n$-vertex fan-planar multigraphs with $4n - 12$ edges. Removing the thick edges in Figs. 4a and 4c gives bipartite $n$-vertex fan-planar graphs with $4n - 16$ edges. ▶

To prove the upper bound, consider a bipartite fan-planar graph $G$ with a fixed fan-planar drawing. W.l.o.g. assume that $G$ is edge-maximal and connected, and $A, B$ are the two bipartitions of $G$. We shall denote vertices in $A$ by $a, a', a_i$ for some index $i$, and similarly vertices in $B$ by $b, b', b_i$. By fan-planarity, for each crossed edge $e$ of $G$ all edges crossing $e$ have a common endpoint $v$. We call $e$ an $A$-edge (respectively, $B$-edge) if this vertex $v$ lies in $A$ (respectively, $B$). If $e$ is crossed exactly once, it is an $A$-edge and $B$-edge.

A cell of some subgraph $H$ of $G$ is a connected component $c$ of the plane after removing all vertices and edges in $H$; see also [30]. The size of $c$, denoted by $||c||$, is the total number of vertices and edge segments on the boundary of $c$, counted with multiplicities.

Lemma 7 ([30]). Each fan-planar graph $G$ admits a fan-planar drawing such that if $c$ is a cell of any subgraph of $G$, and $||c|| = 4$, then $c$ contains no vertex of $G$ in its interior.

We choose a fan-planar drawing of $G$ with the property given in Lemma 7.

Corollary 8. If $e = (a, b)$, with $a \in A$ and $b \in B$, is crossed at point $p$ by an $A$-edge $e'$, then each edge crossing $e$ between $a$ and $p$ is an $A$-edge crossed by each edge crossing $e'$.

Proof. Let $x$ be the common endpoint of all edges crossing $e$ and $e' = (x, y)$ be the $A$-edge crossing $e$ in $p$. Let $e'' = (x, y')$ be an edge that crosses $e$ between $p$ and $a$. If $e''$ is not an $A$-edge, it is crossed by an edge $e_1 = (a', b)$ with $a' \neq a$. The $A$-edge $e'$ is not crossed by $e_1$. But then there is a cell $c_1$ bounded by vertex $b$ and segments of $e, e''$ and $e_1$, which contains vertex $x$ or $y$ in its interior (see Fig. 5a), contradicting Lemma 7. Symmetrically, if there is
an edge $e_2 = (a, b')$ that crosses $e'$ but not $e''$, then there is a similar cell $c_2$ with $|c_2| = 4$ containing vertex $x$ or $y'$ (see Fig. 5b), again contradicting Lemma 7.

Kaufmann and Ueckerdt [30] derive Lemma 7 from the following lemma.

Lemma 9 ([30]). Let $G$ be given with a fan-planar drawing. If two edges $(v, w)$ and $(u, x)$ cross in a point $p$, no edge at $v$ crosses $(u, x)$ between $p$ and $u$, and no edge at $x$ crosses $(v, w)$ between $p$ and $w$, then $u$ and $w$ are on the boundary of the same cell of $G$.

By the maximality of $G$ we have in this case that $(u, w)$ is an edge of $G$, provided $u$ and $w$ lie in distinct bipartition classes. We can use this fact to derive the following lemma.

Lemma 10. Assume that $e_1 = (a_1, b_1)$ and $e_2 = (a_2, b_2)$ cross. If $e_1$ and $e_2$ are both $A$- or $B$-edges, then $(a_2, b_1)$ belongs to $G$ and can be drawn so that each edge that crosses $(a_2, b_1)$ also crosses $e_2$. Otherwise, $(a_2, b_1)$ belongs to $G$ and can be drawn crossing-free.

Proof. First assume that $e_1$ and $e_2$ are both $A$-edges; the case where $e_1$ and $e_2$ are both $B$-edges is analogous. Let $p_1$ be the crossing point on $e_1$ that is closest to $b_1$. Since $e_1$ is an $A$-edge crossing $(a_2, b_2)$, the edge $e$ crossing $e_1$ at $p_1$ (possibly $e = e_2$) is incident to $a_2$. Now either $e = e_2$ or the subgraph $H$ of $G$ consisting of $e, e_1$ and $e_2$ (and their vertices) has one bounded cell $c$ of size 4, which by Lemma 7 contains no vertex. In both cases, every edge of $G$ crossing $e$ between $a_2$ and $p_1$, also crosses $e_2$ and ends at $a_1$ (as $e_2$ is an $A$-edge crossing $(a_1, b_1)$). Thus, drawing an edge from $b_1$ along $e_1$ to $p_1$ and then along $e$ to $a_2$ does not violate fan-planarity; see Fig. 5c. By the maximality of $G$, $(a_2, b_1)$ belongs to $G$.

Now assume that $e_1$ is an $A$-edge and $e_2$ is a $B$-edge. Let $p$ be the crossing point of $e_1$ and $e_2$. By Lemma 9, $a_2$ and $b_1$ lie on the same cell in $G$ and hence, by the maximality of $G$, we have that the edge $(a_2, b_1)$ is contained in $G$ and can be drawn crossing-free.

We are now ready to prove the main theorem of this section (see also [9] for omitted parts).

Theorem 11. Any $n$-vertex bipartite fan-planar graph has at most $4n - 12$ edges.

Proof sketch. We start by considering the planar structure $G_p$ of $G$, i.e., an inclusion-maximal subgraph of $G$ whose drawing inherited from $G$ is crossing-free. Let $E_A$ and $E_B$ be the set of all $A$-edges and $B$-edges, respectively, in $E[G] - E[G_p]$. Each $e \in E_A$ is crossed by a non-empty (by maximality of $G_p$) set of edges in $G$ with common endpoint $a \in A$, and we say that $e$ charges $a$. Similarly, every $e \in E_B$ charges a unique vertex $b \in B$.

For any vertex $v$ in $G$, let $ch(v)$ denote the number of edges in $E_A \cup E_B$ charging $v$. Moreover, for a multigraph $H$ containing $v$, let $\deg_H(v)$ denote the degree of $v$ in $H$, i.e., the number of edges of $H$ incident to $v$. Our goal is to show that for every vertex $v$ of $G$ we have $\deg_{G_p}(v) - ch(v) \geq 2$. However, this is not necessarily true when $G_p$ is not connected or has faces of length 6 or more. To overcome this issue, we shall add in a step-by-step procedure vertices and edges (possibly parallel but non-homotopic to existing edges in $G_p$) to the plane drawing of $G_p$ such that:
To find \( \bar{G}_p \) (refer to [9] for a full proof), we first assume that \( G_p \) is not connected. In this case there must be an edge \( e \) with endpoints in different connected components of \( G_p \), which is crossed by some edge \( e' \) in \( G_p \). Depending on which of \( e, e' \) is an \( A \)-edge or \( B \)-edge, we either use Lemma 10 to add a new edge to \( G_p \) or we carefully add a new vertex of degree three to \( G_p \). Once we may assume that \( G_p \) is connected but not a quadrangulation, there exists a face \( f \) whose facial walk \( W \) has length at least 6. For edges \( e \) with one endpoint in \( V[W] \) that run through face \( f \) and leave \( f \) by crossing an edge \( e' \) of \( G_p \), we define a stick to be the initial segment of \( e \) that is contained in \( f \). Such a stick \( s \) splits \( W \) into two parts, each going from the start vertex of \( e \) to the crossing of \( e \) and \( e' \). As \( G \) is bipartite, exactly one part, the inner side of \( s \), contains an even number of vertices, and \( s \) is called short if its inner side has only two vertices, and long otherwise. In case \( f \) has a long stick, then again depending on which of \( e, e' \) is an \( A \)-edge or \( B \)-edge, we either use Lemma 10 to add a new edge to \( G_p \) (see Fig. 6a) or we carefully add a new vertex of degree three to \( G_p \) (see Fig. 6b). Finally, if all sticks are short, we can add a crossing-free edge to \( G_p \), or a new vertex with three crossing-free edges to \( G_p \), as shown in Fig. 6c.

Adding to \( G_p \) an edge or a vertex with three edges, strictly increases the average degree in \( G_p \). Hence, we ultimately obtain supergraphs \( \bar{G} \) of \( G \) and \( \bar{G}_p \) of \( G_p \) satisfying P.1–P.3. Next, we show that the charge of every original vertex \( v \) is at most its degree in \( \bar{G}_p \) minus 2.

**Claim 12.** Every \( v \in V[\bar{G}] \) satisfies \( \text{deg}_{\bar{G}_p}(v) - \text{ch}(v) \geq 2 \).

**Proof.** W.l.o.g. consider any \( a \in A \) and let \( k := \text{deg}_{\bar{G}_p}(a) \) and \( S \subseteq E_A \) be the set of edges charging \( a \). Observe that no two edges of \( S \) can cross. In fact, if \((a_1, b_1) \in E_A \) charges \( a \) and \((a_2, b_2) \in E_A \) crosses \((a_1, b_1) \), then \((a_2, b_2) \) charges \( a_1 \neq a \). Consider the face \( f \) of \( \bar{G}_p - \{a\} \) containing \( a \), and the closed facial walk \( W \) around \( f \). Walk \( W \) has length \( 2k \) (counting with repetitions) as \( \bar{G}_p \) is a quadrangulation. Further, each edge in \( S \) lies in \( f \) and has both endpoints on \( W \). Hence, the subgraph of \( \bar{G} \) consisting of all edges in \( W \cup S \) is crossing-free and has vertex set \( V[W] \). Define graph \( J \) by breaking the repetitions along \( W \), i.e., \( J \) consists of a cycle of length \( 2k \) and every edge in \( S \) is an uncrossed chord of this cycle. \( J \) has \( k - 2 \) chords, as it is bipartite outerplanar. Thus, \(|S| = \text{ch}(a) \leq k - 2 = \text{deg}_{\bar{G}_p}(a) - 2 \). Let \( X = V[\bar{G}] - V[\bar{G}_p] \) be the set of newly added vertices. For each \( x \in X \), \( \text{deg}_{\bar{G}_p}(x) \geq 3 \) and \( \text{ch}(x) = 0 \) hold. Thus, \( \text{deg}_{\bar{G}_p}(x) - \text{ch}(x) \geq 3 \), and by Claim 12 we get \( 2|E[\bar{G}_p]| - |E_A| + |E_B| = \sum_{v \in V[\bar{G}_p]} \left( \text{deg}_{\bar{G}_p}(v) - \text{ch}(v) \right) \geq 2n + 3|X| \) which implies \( |E_A| + |E_B| \leq 2|E[\bar{G}_p]| - 2n - 3|X| \). On the other hand, \( |E[G_p]| + 3|X| \leq |E[\bar{G}_p]| \) by P.3 and \( |E[G_p]| = 2(n + |X|) - 4 \) by P.1, which together give \( |E[\bar{G}]| = |E[G_p]| + |E_A| + |E_B| \leq 3|E[\bar{G}_p]| - 6|X| - 2n = 4n - 1 \).
Figure 7 Constructions for dense bipartite $n$-vertex 2-planar (a) graphs and (b) multigraphs.

Figure 8 Illustration of sticks, scissors and twins.

5 Bipartite 2-Planar Graphs

In this section, we overview our result for bipartite 2-planar graphs. For reasons of space, we sketch the proof; the full version is in [9]. We start with the lower bound; see Fig. 7.

Theorem 13. For infinitely many values of $n$, there exists a bipartite $n$-vertex 2-planar
(i) graph with $3.5n - 12$ edges, and
(ii) multigraph with $3.5n - 8$ edges.

For the upper bound, we study structural properties of the planar structure $G_p$ of an optimal bipartite 2-planar graph $G$. Let $(u, v)$ be an edge of $G$ that does not belong to $G_p$. By the maximality of $G_p$, $(u, v)$ has at least one crossing with an edge of $G_p$. As already mentioned, the part of $(u, v)$ that starts from $u$ ($v$) and ends at the first intersection point of $(u, v)$ with an edge of $G_p$ is a stick of $u$ ($v$). When $(u, v)$ has two crossings, there is a part that is not a stick, called middle-part. Each stick or middle-part lies in a face $f$ of $G_p$; we say that $f$ contains this part. Let $f = \langle u_0, u_1, \ldots, u_{k-1} \rangle$ be a face of $G_p$ with $k \geq 4$ and let $s$ be a stick of $u_i$ contained in $f$, $i \in \{0, 1, \ldots, k-1\}$. We call $s$ a short stick, if it ends either at $(u_{i+1}, u_{i+2})$ or at $(u_{i-1}, u_{i-2})$ of $f$; otherwise, $s$ is called a long stick; see Figs. 8a-8b.

W.l.o.g. we assume that among all optimal bipartite $n$-vertex 2-planar graphs, $G$ is such that its planar structure $G_p$ is the densest among the planar structures of all other optimal bipartite $n$-vertex 2-planar graphs; we call $G_p$ maximally dense. We first prove that $G_p$ is a spanning quadrangulation. For this, we first show that $G_p$ is connected, as otherwise it is always possible to augment it by adding an edge joining two connected components of it. Then, we show that all faces of $G_p$ are of length four. Our proof by contradiction is rather technical; assuming that there is a face $f$ with length greater than four in $G_p$, we consider two main cases:
(i) $f$ contains no sticks, but middle-parts, and
(ii) $f$ contains at least one stick.

With a careful case analysis, we lead to a contradiction either to the maximality of $G_p$ or to the fact that $G$ is optimal.

Since $G_p$ is a quadrangulation, it has exactly $2n - 4$ edges and $n - 2$ faces. Our goal is to prove that the average number of sticks for a face is at most 3. Since the number of edges of $G \setminus G_p$ equals half the number of sticks over all faces of $G_p$, this implies that $G$ cannot have more than $2n - 4 + \frac{3}{2}(n - 2) = 3.5n - 7$ edges, which gives the desired upper bound.
Let $f$ be a face of $G_p$. Denote by $h(f)$ the number of sticks contained in $f$. A scissor of $f$ is a pair of crossing sticks starting from non-adjacent vertices of $f$, while a twin of $f$ is a pair of sticks starting from the same vertex of $f$ crossing the same boundary edge of $f$; see Fig. 8c. We refer to a pair of crossing sticks starting from adjacent vertices of $f$ as a pseudo-scissor; see Fig. 8d. The following lemma shows that a face of $G_p$ contains a maximum number of sticks (that is, 4) only in the presence of scissors or twins, due to 2-planarity; see [9].

**Lemma 14.** Let $G$ be an optimal bipartite 2-planar graph, such that its planar structure $G_p$ is maximally dense. Then, for each face $f$ of $G_p$, it holds $h(f) \leq 4$. Further, if $h(f) = 4$, then $f$ contains one of the following: two scissors, or two twins, or a scissor and a twin.

An immediate consequence of Lemma 14 is that $h(f) \leq 3$, for every face $f$ containing a pseudo-scissor. We now consider specific “neighboring” faces of a face $f$ of $G_p$ with four sticks and prove that they cannot contain so many sticks. Observe that each edge corresponding to a stick of $f$ starts from a vertex of $f$ and ends at a vertex of another face of $G_p$. We call this other face, a neighbor of this stick. The set of neighbors of the sticks forming a scissor (twin) of $f$ form the so-called neighbors of this scissor (twin).

By Lemma 14 and since $h(f) = 4$, face $f$ contains two sticks $s_1$ and $s_2$ forming a twin or a scissor, with neighbors $f_1$ and $f_2$. By 2-planarity and based on a technical case analysis, we show that $h(f_1) + h(f_2) \leq 7$ except for a single case, called 8-sticks configuration and illustrated in Fig. 9a, for which $h(f_1) + h(f_2) = 8$.

Assume first that $G$ does not contain any 8-sticks configuration. Let $H$ be an auxiliary graph, called dependency graph, having a vertex for each face of $G_p$. Then, for each face $f$ of $G_p$ containing a scissor or a twin with neighbors $f_1$ and $f_2$, s.t. $h(f_1) \leq h(f_2)$, there is an edge from $f$ to $f_1$ in $H$: $f_1 = f_2$ is possible. To prove that the average number of sticks for a face of $G_p$ is at most 3 (which implies the upper bound), it suffices to prove that the number of faces of $G_p$ that contain two sticks is at least as large as the number of faces that contain four sticks. This holds due to the following facts for every face $f$ of $G_p$:

(i) if $h(f) = 4$, then $f$ has two outgoing edges and no incoming edge in $H$,
(ii) if $h(f) = 3$, then the number of outgoing edges of $f$ in $H$ is at least as large as the number of its incoming edges, and
(iii) if $h(f) = 2$, then $f$ has at most two incoming edges in $H$.

So, $G$ has at most $3.5n - 7$ edges in the absence of 8-sticks configurations.

Finally, if $G$ contains 8-sticks configurations, we eliminate each of them (without creating new) by adding one vertex, and by replacing two edges of $G$ by six other edges violating neither bipartiteness nor 2-planarity, as in Fig. 9b. The derived graph $G'$ has a planar structure that is a spanning quadrangulation without 8-sticks configurations. Since $G'$ has one vertex and four edges more than $G$ for each 8-sticks configuration and since the vertices of $G'$ have degree at most 3.5 on average, by reversing the augmentation steps we conclude that $G$ cannot be denser than $G'$. We summarize our result in the following.
Theorem 15. A bipartite $n$-vertex 2-planar multigraph has at most $3.5n - 7$ edges.

Implications of Theorem 15. In the following, we adjust the well-known Crossing Lemma to bipartite graphs and use it to obtain a bound on the density of bipartite $k$-planar graphs, when $k > 2$. Our proofs are inspired by the ones for general graphs; see, e.g., [4].

Theorem 16. Let $G$ be a bipartite topological graph with $n \geq 3$ vertices and $m \geq \frac{17}{4}n$ edges. Then, $cr(G) \geq \frac{16}{289} \cdot \frac{m^3}{n^2} \approx \frac{1}{114} \cdot \frac{m^3}{n^2}$, where $cr(G)$ is the crossing number of $G$.

Proof. We first prove a weaker bound which holds for every $m$. That is, $cr(G) \geq 3m - \frac{17}{2}n + 19$. This bound clearly holds when $m \leq 2n - 4$. Hence, we may assume w.l.o.g. that $m > 2n - 4$.

It follows from [18] that if $m > 3n - 8$, then $G$ has an edge that is crossed by at least two other edges. Also, by Theorem 15 we know that if $m > \frac{7}{2}n - 7$, then $G$ has an edge that is crossed by at least three other edges. We obtain by induction on the number of edges of $G$ that $cr(G) \geq (m - (2n - 4)) + (m - (3n - 8)) + (m - (\frac{7}{2}n - 7)) = 3m - \frac{17}{2}n + 19$.

Assume that $G$ admits a drawing on the plane with $cr(G)$ crossings and let $p = \frac{17n}{4m} \leq 1$. Choose independently every vertex of $G$ with probability $p$, and denote by $H_p$ the graph induced by the chosen vertices. Let also $n_p$, $m_p$ and $c_p$ be the random variables corresponding to the number of vertices, of edges and of crossings of $H_p$. Taking expectations on the relationship $c_p \geq 3m_p - \frac{17}{2}n_p + 19$, which holds by our weaker bound, we obtain that $p^4cr(G) \geq 3p^3m - \frac{17}{2}np$, or equivalently that $cr(G) \geq \frac{3p^3}{p^4} - \frac{17}{2p}$. The proof follows by plugging $p = \frac{17n}{4m}$ (which is at most 1 by our assumption) to the last inequality.

Theorem 17. Let $G$ be a bipartite $k$-planar graph with $n \geq 3$ vertices and $m$ edges, for some $k \geq 1$. Then: $m \leq \frac{17}{8} \sqrt{2kn} \approx 3.005\sqrt{kn}$.

Proof. For $k = 1, 2$, the bounds are weaker than the ones of [18] and of Theorem 15. So, we may assume w.l.o.g. that $k > 2$. We may also assume that $m \geq \frac{17}{2}n$, as otherwise there is nothing to prove. Combining the fact that $G$ is $k$-planar with the bound of Theorem 16 we obtain that $\frac{16}{289} \cdot \frac{m^3}{n^2} \leq cr(G) \leq \frac{1}{4}mk$, which implies that $m \leq \frac{17}{8} \sqrt{2kn} \approx 3.005\sqrt{kn}$.

Conclusions and Open Problems

We presented tight bounds for the density of bipartite beyond-planar graphs, yielding an improvement of the leading constant of the Crossing Lemma for bipartite graphs. We conclude with open problems.

(i) What is the maximum density of bipartite $k$-planar graphs with $k > 2$? Such bounds may further improve the leading constant of the Crossing Lemma for bipartite graphs; Fig. 9c shows a bipartite 3-planar graph with $4n - O(1)$ edges. Bounds for other classes of bipartite beyond-planar (e.g., quasi-planar) graphs are also interesting.

(ii) The ratio of the maximum density of general over bipartite graphs for large $n$ approaches $\frac{3n}{2m} = 1.5$ for planar graphs, $\frac{4n}{5m} \approx 1.33$ for 1-planar graphs, $\frac{5n}{7m} \approx 1.43$ for 2-planar graphs and at most $\frac{5.5n}{10m} \approx 1.37$ for 3-planar graphs, leaving room for speculation on how it develops for $k$-planar graphs with $k > 3$; note that for classes closed under subgraphs, it is at most 2 [23].

(iii) Optimal 1-, 2- and 3-planar graphs allow for characterizations [13, 35], while recognizing general beyond-planar graphs is often NP-hard. Does the restriction of bipartiteness allow for characterizations or efficient recognition algorithms in some cases?

(iv) Finally, one should study properties that not only hold for general beyond-planar graphs but also for bipartite ones, e.g., is every optimal bipartite RAC graph also 1-planar?
Beyond-Planarity: Turán-Type Results for Non-Planar Bipartite Graphs

References


A Dichotomy Result for Cyclic-Order Traversing Games

Yen-Ting Chen  Department of Computer Science, National Chiao Tung University, Hsinchu City, Taiwan
yentingchen.cs02@nctu.edu.tw

Meng-Tsung Tsai1  Department of Computer Science, National Chiao Tung University, Hsinchu City, Taiwan
mtsai@cs.nctu.edu.tw

Shi-Chun Tsai2  Department of Computer Science, National Chiao Tung University, Hsinchu City, Taiwan
sctsai@cs.nctu.edu.tw

Abstract
Traversing game is a two-person game played on a connected undirected simple graph with a
source node and a destination node. A pebble is placed on the source node initially and then
moves autonomously according to some rules. Alice is the player who wants to set up rules for
each node to determine where to forward the pebble while the pebble reaches the node, so that
the pebble can reach the destination node. Bob is the second player who tries to deter Alice’s
effort by removing edges. Given access to Alice’s rules, Bob can remove as many edges as he
likes, while retaining the source and destination nodes connected. Under the guide of Alice’s
rules, if the pebble arrives at the destination node, then we say Alice wins the traversing game;
otherwise the pebble enters an endless loop without passing through the destination node, then
Bob wins. We assume that Alice and Bob both play optimally.

We study the problem: When will Alice have a winning strategy? This actually models a
routing recovery problem in Software Defined Networking in which some links may be broken. In
this paper, we prove a dichotomy result for certain traversing games, called cyclic-order traversing
games. We also give a linear-time algorithm to find the corresponding winning strategy, if
one exists.

2012 ACM Subject Classification  Mathematics of computing → Graph theory, Theory of com-
putation → Design and analysis of algorithms, Networks → Network reliability

Keywords and phrases  st-planar graphs, biconnectivity, fault-tolerant routing algorithms, soft-
ware defined network

Digital Object Identifier  10.4230/LIPIcs.ISAAC.2018.29

1 Introduction
Several mathematical models were proposed to forward packets in a routing network G,
such as the sink-tree routing model [9, 4, 7], v-acorn routing model [1], and the cyclic-order
routing model [17]. The cyclic-order routing model is fault-tolerant in the sense that packets

1 This research was supported in part by the Ministry of Science and Technology of Taiwan under contract
MOST grant 107-2218-E-009-026-MY3, and the Higher Education Sprout Project of National Chiao
Tung University and Ministry of Education (MOE), Taiwan.

2 This research was supported in part by the Ministry of Science and Technology of Taiwan under contracts
MOST 105-2622-8-009-008 and 105-2221-E-009-103-MY3.
Algorithm 1: The pebble-moving algorithm.

Input: A connected undirected simple graph \( G = (V \cup \{s, t\}, E) \),
Alice’s strategy \( \{\pi_{v \to x}, \pi_{x \to v} : (v, x) \in E\} \),
Bob’s removal of edges \( E_B \) so that \( G - E_B \) remains st-connected.

1. \( \text{pre} \leftarrow s \), \( \text{cur} \leftarrow s \);
2. while \( \text{cur} \neq t \) do
3.    foreach \( (\text{cur}, u) \) in ordered list \( \pi_{\text{pre} \to \text{cur}} \) do
4.      if \( (\text{cur}, u) \notin E_B \) then
5.        \( \text{pre} \leftarrow \text{cur} \), \( \text{cur} \leftarrow u \);
6.        break;
7.    end
8.  end
9. return “Alice wins”;
that may only support matching and forwarding packets. Software Defined Networking has been a focus in network and communication research in recent years, since McKeown et al. published their pioneering work [18]. Because an SDN controller needs to monitor multiple OpenFlow switches and constant interaction between controller and switches may slow down the network, some fast failover mechanism is devised [19], in particular the group table used in the OpenFlow protocol. When a packet enters an OpenFlow switch, the flow rules will match related fields in the packet to determine from which port the packet enters and then go to the corresponding group table. Each group table has a bucket list to watch whether the links are up or down. The bucket lists relate to the ordered lists $\pi_{v \rightarrow x}$ for each ordered pair of nodes $v \rightarrow x$ in the traversing game. When a link is down, the switch can quickly select the next bucket in the link’s failover group table with a watch port that is up. It thus can be used to reduce the interaction between controllers and switches when link failures are detected, which is an important issue studied in SDN [19]. The traversing game is an abstraction of the above protocol.

Deciding who wins the traversing game is a problem in $\Sigma_2^p$ [8, 2], which is the set of all languages $L$ for which there exists a polynomial time Turing machine $M$ and a polynomial $q$ such that $x \in L \iff \exists u \in \{0, 1\}^{q(|x|)} \forall y \in \{0, 1\}^{q(|x|)} M(x, u, y) = 1$. We do not know whether it can be solved in polynomial time, even with a nondeterministic Turing machine. We thus impose a restriction on all the ordered lists $\pi_{v \rightarrow x}$ in the traversing game, which makes the traversing game solvable in linear time. Let $\pi_x$ be a cyclic order of the edges incident to $x$. The restriction is, for each node $x \in V \cup \{s, t\}$, there exists a cyclic order $\pi_x$ so that for every ordered pair of nodes $v \rightarrow x$, the ordered list $\pi_{v \rightarrow x}$ is equal to the segment of $\pi_x$ that starts from the successor of $(x, v)$ and finishes at $(x, v)$. We say a traversing game with the above restriction cyclic-order traversing game. In [17], the authors show that Alice has a winning strategy for a cyclic-order traversing game if the underlying graph $G$ is comprised of (hierarchical) node-disjoint paths. We will show how to generalize this finding.

We need some notions to state our main result. $st$-planar graphs were first introduced by Lempel et al. [16], which are acyclic planar digraphs with exactly one source node $s$ and exactly one sink node $t$ and can be embedded in the plane so that $s, t$ are both on the outer face. This definition was later adapted, for example in [3], to be such undirected graphs that have a planar embedding with $s$ and $t$ on the same face, or equivalently both on the outer face. We use the latter definition of $st$-planar graphs in this paper.

The $st$-biconnected component $B_{st}(G)$ of an undirected graph $G$ is defined to be the subgraph of $G$ induced by the nodes in the biconnected component of $G \cup \{(s, t)\}$ that contains $(s, t)$. Or equivalently, as shown in Lemma 2, $B_{st}(G)$ is the node-induced subgraph of $G$ with the removal of all the nodes that are not on any simple path in $G$ from node $s$ to node $t$, i.e. ignorable nodes. It is clear that the removal of ignorable nodes cannot make the status of other nodes changed from unignorable to ignorable, so $B_{st}(G)$ is a unique subgraph of $G$, regardless of the sequence of node removals.

Our main result is:

\textbf{Theorem 1.} For a cyclic-order traversing game with underlying graph $G = (V \cup \{s, t\}, E)$, Alice has a winning strategy if and only if $B_{st}(G)$ is $st$-planar. In addition, there exists an $O(|V| + |E|)$-time algorithm that either outputs Alice’s winning strategy or determines that there is none.
Related Work
Annexstein et al. [1] also proposed a mathematical model for fault-tolerant routing. In their routing scheme, they need to assign an acyclic orientation to the underlying graph \( G = (V \cup \{s, t\}, E) \) so that every node other than \( t \) (the sink node) has at least \( k \) out-going directed edges and \( k \) is the maximum possible among all acyclic orientations. Given the acyclic orientation, packets at node \( x \) are forwarded to any available out-going edge of \( x \). As long as \( t \) is functioning and fewer than \( k \) nodes malfunction, packets can arrive at \( t \) from nodes other than \( t \). The orientation can be found in linear time.

The routing scheme by Annexstein et al. has no re-routing, so it is efficient to forward packets. It is clear that our routing scheme covers all the cases that Annexstein et al.’s model can handle if the \( st \)-biconnected component \( B_{st}(G) \) of the underlying graph \( G \) is \( st \)-planar, so it is more fault-tolerant for such graphs, in tradeoff of the cost to re-route packets. Our routing strategy can be found in linear time as well.

Organization
The rest of the paper is organized as follows. In Section 2, we show some graph properties for \( st \)-biconnected components, which are used as building blocks for the proofs in the subsequent sections. In Section 3, we show that Alice has a winning strategy for any cyclic-order traversing game when the \( B_{st}(G) \) of the underlying graph \( G = (V \cup \{s, t\}) \) is \( st \)-planar. In Section 4, we show that the graph class studied in Section 3 is the exact graph class that Alice has a winning strategy for cyclic-order traversing games, by studying the situations that Bob has a winning strategy. Finally, in Section 5, a linear-time algorithm is given to compute Alice’s winning strategy, if one exists.

2 Properties of \( st \)-Biconnected Components
In this section, we show some properties of \( st \)-biconnected components, which are used as building blocks for the proofs in subsequent sections. Here are some notations to simplify the presentation. By \( G - \{x\} \) (resp. \( G - \{(x, y)\} \)), we denote to remove node \( x \) (resp. edge \( (x, y) \)) from \( G \). By \( G \cup \{(x, y)\} \), we denote to add an edge \( (x, y) \), if not existing, to \( G \). Let \( st \)-path denote an undirected path from \( s \) to \( t \). We say a graph is \( st \)-connected if it has an \( st \)-path.

We begin with a proof showing that the two definitions of \( B_{st}(G) \) are equivalent.

\[ \blacktriangleright \textbf{Lemma 2.} \text{ For every connected undirected simple graph } G = (V \cup \{s, t\}, E), \text{ removing all nodes that are not on any simple } st \text{-path in } G \text{ yields } B_{st}(G). \]

\[ \textbf{Proof.} \] Let \( C \) be the graph obtained by removing all nodes that are not on any simple \( st \)-path from \( G \). Since \( G \) is connected, \( s \) and \( t \) are on a simple \( st \)-path, so \( s, t \in C \). On the other hand, \( s, t \in B_{st}(G) \) by the definition of \( st \)-biconnected component. We need to discuss for those nodes other than \( s \) and \( t \).

Let \( v \) be a node in \( B_{st}(G) \) other than \( s, t \). Since \( B_{st}(G) \cup \{(s, t)\} \) is biconnected, there are two node-disjoint paths from \( \{v\} \) to \( \{s, t\} \) in \( B_{st}(G) \cup \{(s, t)\} \) that have only node \( v \) in common by Menger’s Theorem [14]. Joining these two paths gives a simple path from \( s \) to \( t \) that passes through \( v \), so \( v \) is a node in \( C \).

Let \( v \) be a node in \( C \) other than \( s, t \). Then there is a simple \( st \)-path that passes through \( v \). Together with the edge \( (s, t) \), this gives a simple cycle containing \( s, v, t, \) so \( v \) is a node in \( B_{st}(G) \). \[ \blacktriangleright \]
By Lemma 2 and the properties of block-cut trees [10, 13], we get:

**Corollary 3.** For every node $v$ in a connected undirected simple graph $G = (V \cup \{s, t\}, E)$, $v$ is not contained in $B_{st}(G)$ if and only if $v$ can be disconnected from $s$ and $t$ by removing an articulation point $x$ where $x \in B_{st}(G)$ and $x \neq v$.

**Lemma 4.** Given a connected undirected simple graph $G = (V \cup \{s, t\}, E)$, let $H$ be any subgraph of $B_{st}(G)$ so that $H$ has at least two nodes and one edge, then there exists a simple st-path in $B_{st}(G)$ that contains at least two nodes in $H$.

Proof. If both $s$ and $t$ are in $H$, then any simple path $P$ in $B_{st}(G)$ from $s$ to $t$ contains at least two nodes in $H$, e.g. $s$ and $t$. Such a simple path $P$ must exist because $G$ is connected.

If precisely one of $s, t$ is in $H$, then $H$ has a node $v \notin \{s, t\}$. By the definition of st-biconnected component, there exists a simple path $P$ in $B_{st}(G)$ from $s$ to $t$ that passes through $v$. Hence, $P$ contains at least two nodes in $H$. 

The remaining case happens when none of $s$ and $t$ is in $H$, so $H$ has an edge $(u, v)$ where $u, v \notin \{s, t\}$. Let $V_1$ be the node set $\{u, v\}$ and $V_2$ be the node set $\{s, t\}$. By the definition of st-biconnected component, there is a simple path $P_u$ (resp. $P_v$) in $B_{st}(G)$ from $s$ to $t$ that passes through $u$ (resp. $v$). In the subgraph $P_u \cup P_v$, to disconnect $u$ (resp. $v$) from $V_2$ by removing a single node, $u$ (resp. $v$) must be the node to be removed. Since $u \neq v$, one cannot disconnect $V_1$ from $V_2$ by removing a single node. Thus by Menger’s Theorem, there are two node-disjoint paths $P_1, P_2$ from $V_1$ to $V_2$. Joining $P_1, P_2$ with $(u, v)$ yields the desired path.

**Lemma 5.** For any cyclic-order traversing game, if the pebble-moving algorithm does not stop, i.e. the pebble does not arrive $t$, then every possible move $v \rightarrow x$ either does not occur or occur more than once.

Proof. If the pebble-moving algorithm does not stop, then the tour of the pebble can be represented as an infinite sequence of moves $v_1 \rightarrow x_1, v_2 \rightarrow x_2, \ldots$. Let $S_i$ be the subsequence of the moves after $v_i \rightarrow x_i$, and let $j$ be the smallest $j \geq i$ so that $v_j \rightarrow x_j$ occurs more than once in $S_i$. Since $v_j \rightarrow x_j$ occurs in $S_i$, let $S_i$ be the smallest, because $S_i$ is an infinite sequence of moves and the number of different moves is finite. We claim that $j = i$. Here is why. Suppose $j > i$, then $v_{j-1} \rightarrow x_{j-1}$ is a move repeated in $S_i$, because in a cyclic-order traversing game the predecessor moves of each occurrence of $v_j \rightarrow x_j$ are the same, yielding a contradiction. Therefore $v_i \rightarrow x_j$ is the first move repeats in $S_i$. This fact holds for every single $i \geq 1$, so every move in the tour repeats more than once.

We are ready to show that Alice can create a winning strategy by bypassing ignorable nodes.

**Lemma 6.** Alice has a winning strategy for a cyclic-order traversing game with underlying graph $G = (V \cup \{s, t\}, E)$ if and only if Alice has a winning strategy for a cyclic-order traversing game with underlying graph $B_{st}(G)$.

Proof. ($\Rightarrow$) Let $X = \{\pi_{v \rightarrow x}, \pi_{x \rightarrow v} : (v, x) \in E\}$ be Alice’s winning strategy on $G$. Then it works for all st-connected subgraphs, in particular $B_{st}(G)$. Then we let $X' = X$, and remove all the edges in $E(G) - E(B_{st}(G))$ from these ordered lists in $X'$, and therefore $X'$ is a valid strategy on $B_{st}(G)$. Since $X'$ has the same behavior as $X$ on $B_{st}(G)$ and its st-connected subgraphs, $X'$ is a winning strategy on $B_{st}(G)$.

($\Leftarrow$) We prove by induction on $|V(G)| - |V(B_{st}(G))|$. It is clear that the statement is true when $|V(G)| = |V(B_{st}(G))|$. Assume $|V(G)| - |V(B_{st}(G))| = k$ for some $k \geq 1$, and the statement holds up to $k - 1$. By Corollary 3, there is an articulation point $u$ separating $s, t$
from a node not in $B_{st}(G)$. Let $C$ be the subgraph of $G - u$ obtained by removing all the components of $G - u$ that contains $s$ or $t$. Because of the existence of $u$, all the nodes in $C$ are not in $B_{st}(G)$ and thus $B_{st}(G) = B_{st}(G - C)$. Then by the induction hypothesis, there is a winning strategy $Y = \{\pi_{v\rightarrow x}, \pi_{x\rightarrow y} : (v, x) \in E(G - C)\}$ on $G - C$, by which we construct a winning strategy $Y' = \{\pi'_{v\rightarrow x}, \pi'_{x\rightarrow y} : (v, x) \in E\}$ on $G$. Let $y$ be any neighbor of $u$, and $z$ be the neighbor of $u$ connected by $\pi_{y\rightarrow u}(1)$, i.e. the first edge in the ordered list $\pi_{y\rightarrow u}$. Set $\pi'_{y\rightarrow u}$ as any ordered list of those edges connecting $u$ to $C$ followed by $\pi_{y\rightarrow u}$. For each neighbor $v$ of $u$ other than $y$, set $\pi'_{v\rightarrow u}$ as a circular shift of $\pi'_{y\rightarrow u}$ so that the requirement of cyclic-order strategy is satisfied. For each node $x$ in $B_{st}(G) - \{u\}$ and its neighbor $v$, set $\pi'_{v\rightarrow x}$ as $\pi_{v\rightarrow x}$. In this way, the neighbors of $u$ in $C$ are placed together. If the pebble moves from $y$ to $u$ and then $C$, by Lemma 5 it will eventually leave $C$ and $u$ by traversing the edge $(u, z)$, i.e. it will move from $u$ to $z$ after several steps rather than loop in $C$ endlessly. Therefore, when applying $Y'$ to $G$, the pebble moves exactly the same as applying $Y$ to $G - C$ if we ignore the tour of the pebble outside $G - C$. Finally, to see that $Y'$ is indeed a winning strategy, consider any $st$-connected subgraph $H$ of $G$. Let $P$ be a simple $st$-path on $H$. $P$ does not pass through $C$ and therefore $H - C$ is $st$-connected. Then the pebble moves to $t$ when applying $Y$ to $H - C$, as well as when applying $Y'$ to $H$.

3 Winning Strategies for Alice

In this section, we will show that Alice has a winning strategy for a cyclic-order traversing game with underlying graph $G = (V \cup \{s, t\}, E)$ if $B_{st}(G)$ is $st$-planar; that is, the direction $(\leq)$ in Theorem 1. Surprisingly, Alice has no winning strategy if the underlying graph is outside the above graph class, as shown in Section 4. Hence we get a dichotomy result for cyclic-order traversing games.

We begin with a proof showing a base case that the underlying graph $G = (V \cup \{s, t\}, E)$ is $st$-planar.

Lemma 7. Alice has a winning strategy for a cyclic-order traversing game with underlying graph $G = (V \cup \{s, t\}, E)$ if $G$ is $st$-planar.

Proof. Since $G$ is $st$-planar, one can have a planar embedding for $G$ so that $s$, $t$, $(s, s)$ are on the outer face. Given the planar embedding, for each node $x \in V \cup \{s, t\}$, order the edges incident to $x$ clockwise with respect to $x$, which yields a cyclic order $c_x$. We claim that Alice has a winning strategy by setting $\pi_{x\rightarrow y} = c_x$ for each $x \in V \cup \{s, t\}$. In the pebble-moving algorithm, when the pebble is moved from node $x$ to node $y$, the algorithm searches for the next available edge in $\pi_{x\rightarrow y}$, say $(y, z)$, then the pebble is moved along $(y, z)$. Since we set $\pi_{y\rightarrow y} = c_y$, the transit from $(x, y)$ to $(y, z)$ acts like rotating clockwise with respect to $y$. As noted in [20], such a sequence of moves makes the pebble traverse all the edges on a single face if $G$ is connected. Since the pebble starts the tour from the edge $(s, s)$, an edge on the outer face, it will visit all the nodes on the outer face, in particular $s$ and $t$. We depict the tour of the pebble in Figure 1.
No matter how Bob removes edges from $G$, $s$ and $t$ still stay on the outer face. To see why, imagine that for every point $p$ on the outer face there is a curve from $p$ to infinity without crossing any node or edge in $G$. Clearly, removing any subset of edges in $G$ cannot cut the curve, a certificate that $p$ is on the outer face. This yields that, the pebble always visits $s$ and $t$ after any removal of edges, unless $s$ and $t$ are disconnected. In other words, Alice has a winning strategy when the underlying graph is $st$-planar, as claimed.

Together with Lemma 6, we get:

**Theorem 8.** Alice has a winning strategy for a cyclic-order traversing game with underlying graph $G = (V \cup \{s, t\}, E)$ if $B_{st}(G)$ is $st$-planar.

We remark that, in the proof of Lemma 7, if all nodes in $G$ are on the outer face in the planar embedding, i.e. an outerplanar graph [5], then the pebble will visit all nodes regardless of Bob’s removal of edges. This immediately yields that:

**Corollary 9.** Alice has a fixed winning strategy for a cyclic-order traversing game with underlying graph $G = (V, E)$, if $G$ is outerplanar and for all choices of $s, t \in V$.

## 4 Winning Strategies for Bob

In this section, we will show that Bob has a winning strategy for a cyclic-order traversing game with underlying graph $G = (V \cup \{s, t\}, E)$ if $B_{st}(G)$ is not $st$-planar; that is, the contraposition of the direction ($\Rightarrow$) in Theorem 1. Together with the results shown in Section 3, this gives a dichotomy result for cyclic-order traversing games.

We begin with proofs showing base cases where $G$ is $K_5$, $K_{3,3}$, and their subdivisions.

**Lemma 10.** Bob has a winning strategy for a cyclic-order traversing game with underlying graph $G = (V \cup \{s, t\}, E)$ if $G \cup \{(s, t)\}$ is isomorphic to $K_5$.

**Proof.** We prove the case where $(s, t) \notin E$, then the other case follows. The graphs in Figure 2 are possible subgraphs of $G$ after Bob’s removal of edges. We show that Alice cannot assign an ordered list to each $\pi_{s\to x}$ that simultaneously works for $H_1$, $H_2$, and $H_3$. Hence, Bob has a winning strategy on $G$.

To see why, Alice may set $\pi_{s\to x}$ as any of the following six ordered lists.

- **list** 1: $(s, v_1), (s, v_2), (s, v_3), (s, s)$
- **list** 2: $(s, v_1), (s, v_3), (s, v_2), (s, s)$
- **list** 3: $(s, v_2), (s, v_1), (s, v_3), (s, s)$

3 In Lemmas 10, 11, and 12, we ignore the self-loop $(s, s)$ while deciding graph isomorphism.
Lemma 11. ▶

However, if Alice sets \( \pi_{s \rightarrow s} = \text{list}_2 \), then it does not work for \( H_1 \), because \( \pi_{v_2 \rightarrow s} = (s, s), (s, v_1), (s, v_3), (s, v_2) \) and the pebble moves in the cycle \( s, v_1, v_2, v_3, s \) without passing through \( t \). Moreover, if Alice sets \( \pi_{s \rightarrow s} = \text{list}_4 \), then it also does not work for \( H_1 \), because the pebble moves in the cycle \( s, v_1, v_2, v_3, s \). By the same argument, one can show that setting \( \pi_{s \rightarrow s} = \text{list}_1 \) or \( \text{list}_6 \) does not work for \( H_2 \), and setting \( \pi_{s \rightarrow s} = \text{list}_3 \) or \( \text{list}_5 \) does not work for \( H_3 \). This already excludes all possibilities, thus completing the proof. ▶

Lemma 11. Bob has a winning strategy for a cyclic-order traversing game with underlying graph \( G = (V \cup \{s, t\}, E) \) if \( G \) or \( G \cup \{s, t\} \) is isomorphic to \( K_{3,3} \).

Proof. First we consider the case where \( s \) and \( t \) are in the same partition. The graphs in Figure 3 are possible subgraphs of \( G \) after Bob’s removal of edges. Alice may set \( \pi_{s \rightarrow s} \) as any of the following six ordered lists.

\[
\begin{align*}
\text{list}_1 & : (s, v_2), (s, v_3), (s, v_1), (s, s) \\
\text{list}_2 & : (s, v_2), (s, v_4), (s, v_3), (s, s) \\
\text{list}_3 & : (s, v_1), (s, v_2), (s, v_4), (s, s) \\
\text{list}_4 & : (s, v_1), (s, v_3), (s, v_2), (s, s) \\
\text{list}_5 & : (s, v_4), (s, v_2), (s, v_3), (s, s) \\
\text{list}_6 & : (s, v_4), (s, v_3), (s, v_2), (s, s)
\end{align*}
\]

However, if Alice sets \( \pi_{s \rightarrow s} = \text{list}_2 \), then it does not work for \( H_1 \), because \( \pi_{v_2 \rightarrow s} = (s, s), (s, v_1), (s, v_3), (s, v_2) \) and the pebble moves in the cycle \( s, v_1, v_2, v_3, s \) without passing through \( t \). Moreover, if Alice sets \( \pi_{s \rightarrow s} = \text{list}_4 \), then it also does not work for \( H_1 \), because the pebble moves in the cycle \( s, v_1, v_2, v_3, s \). By the same argument, one can show that setting \( \pi_{s \rightarrow s} = \text{list}_1 \) or \( \text{list}_6 \) does not work for \( H_2 \), and setting \( \pi_{s \rightarrow s} = \text{list}_3 \) or \( \text{list}_5 \) does not work for \( H_3 \).
Figure 5 Subdivisions of $K_5$ and $K_{3,3}$.

Next we consider the case where $s$ and $t$ are in different partitions, and prove the subcase where $(s, t) \notin E$, then the other subcase follows. Consider the graphs in Figure 4. Alice may set $\pi_{s \to s}$ as any of the following two ordered lists.

- $\text{list}_1$: $(s, v_3), (s, v_4), (s, s)$
- $\text{list}_2$: $(s, v_1), (s, v_3), (s, s)$

Alice may also set $\pi_{s \to v_3}$ as any of the following two ordered lists.

- $\text{list}_3$: $(v_3, v_1), (v_3, v_2), (v_3, s)$
- $\text{list}_4$: $(v_3, v_2), (v_3, v_1), (v_3, s)$

However, if Alice sets $\pi_{s \to s} = \text{list}_1$ and $\pi_{s \to v_3} = \text{list}_3$, then it does not work for $H_4$, because the pebble moves in the cycle $s, s, v_3, v_1, v_4, s, s$ without passing through $t$. Moreover, if Alice sets $\pi_{s \to s} = \text{list}_2$ and $\pi_{s \to v_3} = \text{list}_4$, then it also does not work for $H_4$, because the pebble moves in the cycle $s, s, v_4, v_1, v_3, s, s$. By the same argument, one can show that setting $\pi_{s \to s} = \text{list}_1$ and $\pi_{s \to v_3} = \text{list}_4$ does not work for $H_5$. Setting $\pi_{s \to s} = \text{list}_2$ and $\pi_{s \to v_3} = \text{list}_4$ also does not work for $H_5$. This already excludes all possibilities, thus completing the proof.

Lemma 12. Bob has a winning strategy for a cyclic-order traversing game with underlying graph $G = (V \cup \{s, t\}, E)$ if $G$ or $G \cup \{(s, t)\}$ is isomorphic to a subdivision of $K_5$ or $K_{3,3}$.

Proof. Bob has a winning strategy if and only if he has one after removing a node with degree one, except $s$ and $t$. This is also true for smoothing out a node with degree two or subdividing an edge, because removing an incident edge of a node with degree two is equivalent to removing both. Therefore we first transform $G$ or $G \cup \{(s, t)\}$ into one of the graphs in Figure 5.

If both $s$ and $t$ belong to $V(K_5)$ or $V(K_{3,3})$, by Lemma 10 and Lemma 11 the statement is true. In what follows, we consider other choices of $s$ and $t$.

Case 1: $G$ or $G \cup \{(s, t)\}$ is isomorphic to $G_1$.

Case 1(a): $\{s, t\} = \{v_1, v_6\}$, or $s = v_6$ and $t = v_7$. Note that $(s, t)$ may or may not belong to $E(G)$. Assume $(s, t) \notin E(G)$, $s = v_6$, and $t = v_1$. By Lemma 10 Alice has no winning strategy to move pebble from $v_2$ to $v_1$, and therefore from $v_6$ to $v_1$. The remaining cases can be reduced to this one.

Case 1(b): $s = v_1$ and $t = v_6$. Note that $G$ is isomorphic to $G_1$ in this case. We first smooth out nodes with degree two, i.e. $v_6$, $v_7$, and $v_9$. Let $D$ be the collection of $v_1v_2$-connected subgraphs of $K_5 - \{(v_1, v_2)\}$. Let $R$ be the collection of $v_1v_8$-connected subgraphs of $G$ that contains $(v_2, v_8)$ but not $(v_1, v_2)$. Define $f : D \to R$ to be the...
29:10 Cyclic-Order Traversing Games

Figure 6 Subgraphs of \( G \) that contain \((v_2, v_8)\). Bijsction from \( D \) to \( R \) such that \( f(D) \in R \) is obtained from \( D \in D \) by adding \((v_2, v_8)\) and replacing \((v_2, v_3)\) with \((v_8, v_3)\) if it is in \( D \). For any \( D \in D \), the pebble moves exactly the same on \( D \) and \( f(D) \). By Lemma 10, for each of Alice’s strategies, the pebble cannot move from \( v_1 \) to \( v_2 \) for some \( D \in D \), and also cannot move from \( v_1 \) to \( v_8 \) for \( f(D) \).

Case 1(c): \( s = v_7 \) and \( t = v_8 \). Similar to the proof of Case 1(b), we let \( R \) be the collection of \( v_7v_8 \)-connected subgraphs of \( G \) that contains \((v_2, v_8)\) and \((v_7, v_1)\), but not \((v_7, v_2)\).

Case 1(d): \( s = v_8 \) and \( t = v_1 \), or \( s = v_8 \) and \( t = v_9 \). Consider the graphs in Figure 6. Alice may set \( \pi_{s \rightarrow s} \) as any of the following two ordered lists.

\[
\text{list}_{14}: (s, v_2), (s, v_1), (s, s)
\]

\[
\text{list}_{15}: (s, v_3), (s, v_2), (s, s)
\]

Alice may also set \( \pi_{s \rightarrow v_2} \) as any of the following two ordered lists.

\[
\text{list}_{16}: (v_2, v_1), (v_2, v_3), (v_2, s)
\]

\[
\text{list}_{17}: (v_2, v_3), (v_2, s), (v_2, s)
\]

However, if Alice sets \( \pi_{s \rightarrow s} = \text{list}_{14} \) and \( \pi_{s \rightarrow v_2} = \text{list}_{16} \), then it does not work for \( H_1 \), because the pebble moves in the cycle \( s, s, v_2, v_4, v_3, s, s \) without passing through \( t \). Moreover, if Alice sets \( \pi_{s \rightarrow s} = \text{list}_{15} \) and \( \pi_{s \rightarrow v_2} = \text{list}_{17} \), then it also does not work for \( H_1 \), because the pebble moves in the cycle \( s, s, v_3, v_4, v_2, s, s \). By the same argument, one can show that setting \( \pi_{s \rightarrow s} = \text{list}_{14} \) and \( \pi_{s \rightarrow v_2} = \text{list}_{16} \) does not work for \( H_2 \), and setting \( \pi_{s \rightarrow s} = \text{list}_{15} \) and \( \pi_{s \rightarrow v_2} = \text{list}_{17} \) also does not work for \( H_2 \).

Case 2: \( G \) or \( G \cup \{(s, t)\} \) is isomorphic to \( G_2 \).

Case 2(a): \( \{s, t\} \in \{(v_1, v_7)\} \) or \( s = v_7 \) and \( t = v_8 \). Assume \( (s, t) \notin E(G) \), \( s = v_7 \), and \( t = v_1 \). By Lemma 11 Alice has no winning strategy to move pebble from \( v_4 \) to \( v_1 \), and therefore from \( v_7 \) to \( v_1 \). The remaining cases can be reduced to this one.

Case 2(b): \( s = v_1 \) and \( t = v_9 \). Similar to the proof of Case 1(b), we let \( R \) be the collection of \( v_1v_9 \)-connected subgraphs of \( G \) that contains \((v_4, v_9)\) but not \((v_1, v_4)\).

Case 2(c): \( s = v_8 \) and \( t = v_9 \). Similar to the proof of Case 1(b), we let \( R \) be the collection of \( v_8v_9 \)-connected subgraphs of \( G \) that contains \((v_4, v_9)\) and \((v_1, v_9)\), but not \((v_4, v_8)\).

Case 2(d): \( s = v_9 \) and \( t = v_1 \), or \( s = v_9 \) and \( t = v_{10} \). Consider the graphs in Figure 7. Alice may set \( \pi_{s \rightarrow s} \) as any of the following two ordered lists.

\[
\text{list}_{18}: (s, v_2), (s, v_4), (s, s)
\]

\[
\text{list}_{19}: (s, v_4), (s, v_2), (s, s)
\]

Alice may also set \( \pi_{s \rightarrow v_2} \) as any of the following two ordered lists.

\[
\text{list}_{20}: (v_2, v_6), (v_2, v_6), (v_2, s)
\]

\[
\text{list}_{21}: (v_2, v_6), (v_2, v_5), (v_2, s)
\]
However, if Alice sets $\pi_{s \rightarrow s} = \text{list}_1$ and $\pi_{s \rightarrow v_2} = \text{list}_4$, then it does not work for $H_3$, because the pebble moves in the cycle $s, s, v_2, v_6, v_3, v_4, s, s$ without passing through $t$. Moreover, if Alice sets $\pi_{s \rightarrow s} = \text{list}_2$ and $\pi_{s \rightarrow v_2} = \text{list}_3$, then it also does not work for $H_3$, because the pebble moves in the cycle $s, s, v_4, v_3, v_6, v_2, s, s$. By the same argument, one can show that setting $\pi_{s \rightarrow s} = \text{list}_1$ and $\pi_{s \rightarrow v_2} = \text{list}_3$ does not work for $H_4$, and setting $\pi_{s \rightarrow s} = \text{list}_2$ and $\pi_{s \rightarrow v_2} = \text{list}_4$ also does not work for $H_4$. ◁

**Theorem 13.** Bob has a winning strategy for a cyclic-order traversing game with underlying graph $G = (V \cup \{s, t\}, E)$ if $B_st(G)$ is not st-planar.

**Proof.** $B_st(G)$ is not st-planar implies that $B_st(G) \cup \{(s, t)\}$ is non-planar. By Kuratowski’s Theorem [15], $B_st(G) \cup \{(s, t)\}$ has a Kuratowski subgraph $H$, i.e. a subdivision of $K_5$ or $K_{3,3}$. By Lemma 4, Bob can find a simple path $P$ from $s$ to $t$ in $B_st(G)$ that passes through at least two nodes in $H$. Let $P_1$ be the subpath starting from $s$ and finishing at the first node in $P$ that is contained in $H$. Let $P_2$ be the subpath starting from the last node in $P$ that is contained in $H$ and finishing at $t$. Bob’s winning strategy is to remove all the edges outside $H - \{(s, t)\} \cup P_1 \cup P_2$. By applying Lemma 12 and Lemma 6, we are done. ◁

## 5 Linear-Time Algorithm

Finally, we give a linear-time algorithm that either outputs Alice’s winning strategy or outputs “Bob wins.” This completes the proof of Theorem 1.

**Theorem 14.** For any cyclic-order traversing game, one can use Algorithm 2 to find a winning strategy for Alice in linear time, if one exists.

**Proof.** By Lemma 2, Step 1 is equivalent to finding the biconnected component in $G \cup \{(s, t)\}$ that contains $s$ and $t$, which can be computed in linear time [11]. By Theorem 8, Step 2, 3, and 4 are equivalent to testing planarity and embedding $B_st(G) \cup \{(s, t)\}$ in the plane, which also can be solved in linear time [6, 12, 21]. For Step 5, the conversion can be done by bypassing ignorable nodes as shown in Lemma 6, which also takes linear time. In total, it takes linear time to find a winning strategy for Alice. ◁

## 6 Conclusion

We identify an interesting traversal problem from a practical network paradigm— software defined networking. We discover that for st-planar graphs we can always find a way in linear time to set up the cyclic-order rules for autonomous re-routing. This can be useful
for designing fault-tolerant network. However, if we allow different type of rules, instead of cyclic ones, then it is not clear when Alice can have a winning strategy. We leave it as an open problem. Meanwhile, we do not know the exact complexity class of the traversing game. We conjecture that it can be $\Sigma^p_2$-complete.

**References**

Y.-T. Chen, M.-T. Tsai, and S.-C. Tsai


The $b$-Matching Problem in Distance-Hereditary Graphs and Beyond

Guillaume Ducoffe
ICI – National Institute for Research and Development in Informatics, Bucharest, Romania
The Research Institute of the University of Bucharest ICUB, Bucharest, Romania
guillaume.ducoffe@ici.ro

Alexandru Popa
University of Bucharest, Bucharest, Romania
ICI – National Institute for Research and Development in Informatics, Bucharest, Romania
alexandru.popa@fmi.unibuc.ro

Abstract
We make progress on the fine-grained complexity of Maximum-Cardinality Matching on graphs of bounded clique-width. Quasi linear-time algorithms for this problem have been recently proposed for the important subclasses of bounded-treewidth graphs (Fomin et al., SODA’17) and graphs of bounded modular-width (Coudert et al., SODA’18). We present such algorithm for bounded split-width graphs – a broad generalization of graphs of bounded modular-width, of which an interesting subclass are the distance-hereditary graphs. Specifically, we solve Maximum-Cardinality Matching in $O((k \log^2 k) \cdot (m+n) \cdot \log n)$-time on graphs with split-width at most $k$. We stress that the existence of such algorithm was not even known for distance-hereditary graphs until our work. Doing so, we improve the state of the art (Dragan, WG’97) and we answer an open question of (Coudert et al., SODA’18). Our work brings more insights on the relationships between matchings and splits, a.k.a., join operations between two vertex-subsets in different connected components. Furthermore, our analysis can be extended to the more general (unit cost) $b$-Matching problem. On the way, we introduce new tools for $b$-Matching and dynamic programming over split decompositions, that can be of independent interest.

2012 ACM Subject Classification Mathematics of computing $\rightarrow$ Graph theory, Theory of computation $\rightarrow$ Design and analysis of algorithms

Keywords and phrases maximum-cardinality matching, $b$-matching, FPT in P, split decomposition, distance-hereditary graphs

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.30

Related Version Full proofs can be found in our technical report [9], https://arxiv.org/abs/1804.09393.

Funding This work was supported by the Institutional research programme PN 1819 “Advanced IT resources to support digital transformation processes in the economy and society - RESINFO-TD” (2018), project PN 1819-01-01 “Modeling, simulation, optimization of complex systems and decision support in new areas of IT&C research”, funded by the Ministry of Research and Innovation, Romania.

1 Introduction

The Maximum-Cardinality Matching problem takes as input a graph $G = (V,E)$ and it asks for a subset $F$ of pairwise disjoint edges of maximum cardinality. This is a fundamental problem with a wide variety of applications. Hence, the computational
complexity of Maximum-Cardinality Matching has been extensively studied in the literature. For instance, this was the first problem shown to be solvable in polynomial-time [11]. Currently, the best-known algorithms for this problem run in $O(m\sqrt{n})$-time on $n$-vertex $m$-edge graphs [22]. Such superlinear running times can be prohibitive for some applications. Intriguingly, Maximum-Cardinality Matching is one of the few remaining fundamental graph problems for which we neither have proved the existence of a quasi linear-time algorithm, nor a superlinear time complexity (conditional) lower-bound. This fact has renewed interest in understanding what kind of graph structure makes this problem difficult. Our present work is at the crossroad of two successful approaches to answer this above question, namely, the quest for improved graph algorithms on special graph classes and the much more recent program of “FPT in P”. We start further motivating these two approaches before we detail our contributions.

1.1 Related work

Algorithmic on special graph classes. One of our initial motivations for this paper was to design a quasi linear-time algorithm for Maximum-Cardinality Matching on distance-hereditary graphs [1]. – Recall that a graph $G$ is called distance-hereditary if the distances in any of its connected induced subgraphs are the same as in $G$. – Distance-hereditary graphs have already been well studied in the literature [1, 8, 17]. In particular, we can solve Diameter in linear-time on this class of graphs [8]. For the latter problem on general graphs, a conditional quadratic lower-bound has been proved in [24]. This result suggests that several hard graph problems in P may become easier on distance-hereditary graphs. Our work takes a new step toward better understanding the algorithmic properties of this class of graphs. We stress that there exist linear-time algorithms for computing a maximum matching on several subclasses of distance-hereditary graphs, such as: trees, cographs [26] and (tent,hexahedron)-free distance-hereditary graphs [7]. However, the techniques used for these three above subclasses are quite different from each other. As a byproduct of our main result, we obtain an $O(m \log n)$-time algorithm for Maximum-Cardinality Matching on distance-hereditary graphs. In doing so, we propose one interesting addition to the list of efficiently solvable special cases for this problem.

Split Decomposition. In order to tackle with Maximum-Cardinality Matching on distance-hereditary graphs, we consider the relationship between this class of graphs and split decomposition. A split is a join that is also an edge-cut. By using pairwise non crossing splits, termed “strong splits”, we can decompose any graph into degenerate and prime subgraphs, that can be organized in a treelike manner. The latter is termed split decomposition [6], and it is our main algorithmic tool for this paper. The split-width of a graph is the largest order of a non degenerate subgraph in some canonical split decomposition. In particular, distance-hereditary graphs are exactly the graphs with split-width at most two [23].

Many NP-hard problems can be solved in polynomial time on bounded split-width graphs (e.g., Graph Coloring, see [23]). Recently, with Coudert, we designed FPT algorithms for polynomial problems when parameterized by split-width [5]. It turns out that many “hard” problems in P such as Diameter can be solved in $O(k^{O(1)} \cdot n + m)$-time on graphs with split-width at most $k$. However, we left this open for Maximum-Cardinality Matching. Indeed, our main contribution in [5] was a Maximum-Cardinality Matching algorithm based on the more restricted modular decomposition. Given this previous result, it was conceivable that a Maximum-Cardinality Matching algorithm based on split decomposition could also exist. However, we need to introduce quite different tools than in [5] in order to prove in this work that it is indeed the case.
Fully Polynomial Parameterized Algorithms. Our work with split-width fits in the recent program of “FPT in P”. Specifically, given a graph invariant denoted $\pi$ (in our case, split-width), we address the question whether there exists a Maximum-Cardinality Matching algorithm running in time $O(k^c \cdot (n + m) \cdot \log^{O(1)}(n))$, for some constant $c$, on every graph $G$ such that $\pi(G) \leq k$. Note that such an algorithm runs in quasi linear time for any constant $k$, and that it is faster than the state-of-the-art algorithm for Maximum-Cardinality Matching whenever $k = O(n^{1-\varepsilon})$, for some $\varepsilon > 0$. This kind of FPT algorithms for polynomial problems have attracted recent attention [5, 16, 19, 20, 21]. We stress that Maximum-Cardinality Matching has been proposed in [21] as the “drosophila” of the study of these FPT algorithms in P. We continue advancing in this research direction.

Note that another far-reaching generalization of distance-hereditary graphs are the graphs of bounded clique-width [17]. In [5], we initiated the complexity study of Maximum-Cardinality Matching – and other graph problems in P – on bounded clique-width graph classes. The latter research direction was also motivated by the recent $O(k^4 \cdot n \log n)$-time algorithm for Maximum-Cardinality Matching on graphs of treewidth at most $k$, see [13, 19]. Turning our attention on denser graph classes of bounded clique-width, we proved in [5] that Maximum-Cardinality Matching can be solved in $O(k^4 \cdot n + m)$-time on graphs with modular-width at most $k$. We stress that distance-hereditary graphs have unbounded treewidth and unbounded modular-width. Furthermore, clique-width is upper-bounded by split-width [23], whereas split-width is upper-bounded by modular-width [5]. As our main contribution in this paper, we present a quasi linear-time algorithm in order to solve some generalization of Maximum-Cardinality Matching on bounded split-width graphs – thereby answering positively to the open question from [5], while improving the state-of-the-art. Our result shows interesting relationships between graph matchings and splits, the latter being an important particular case of the join operation that is used in order to define clique-width. The fine-grained complexity of Maximum-Cardinality Matching parameterized by clique-width, however, remains open.

1.2 Our contributions

We consider a vertex-weighted generalization for Maximum-Cardinality Matching that is known as the unit-cost $b$-Matching problem [12]. Roughly, every vertex $v$ is assigned some input capacity $b_v$, and the goal is to compute edge-weights $(x_e)_{e \in E}$ so that: for every $v \in V$ the sum of the weights of its incident edges does not exceed $b_v$, and $\sum_{e \in E} x_e$ is maximized. We prove a simple combinatorial lemma that essentially states that the cardinality of a maximum $b$-matching in a graph grows as a piecewise linear function in the capacity $b_v$ of any fixed vertex $v$. This nice result (apparently never noticed before) holds for any graph. As such, we think that it could provide a nice tool for the further investigations on $b$-Matching. Then, we derive from our combinatorial lemma a variant of some reduction rule for Maximum-Cardinality Matching that we first introduced in the more restricted case of modular decomposition [5]. Altogether combined, this allows us to reduce the solving of $b$-Matching on the original graph $G$ to solving $b$-Matching on supergraphs of every its split components. We expect our approach to be useful in other matching and flow problems.

Overall, our main result is that $b$-Matching can be solved in $O((k \log^2 k) \cdot (m + n) \cdot \log |b|)$-time on graphs with split-width at most $k$ (Theorem 17). It implies that Maximum-Cardinality Matching can be solved in $O((k \log^2 k) \cdot (m + n) \cdot \log n)$-time on graphs with split-width at most $k$. Since distance-hereditary graphs have split-width at most two, we so obtain the first known quasi linear-time algorithms for Maximum-Cardinality Matching and $b$-Matching on distance-hereditary graphs.
We introduce the required terminology and basic results in Section 2, where we also sketch the main ideas behind our algorithm (Section 2.3). Then, Section 3 is devoted to a combinatorial lemma that is the key technical tool in our subsequent analysis. In Section 4, we present our algorithm for $b$-Matching on bounded split-width graphs. We conclude in Section 5 with some open questions. Due to space restrictions, some of the proofs are omitted. Full proofs can be found in our technical report [9].

2 Preliminaries

We use standard graph terminology from [3]. Graphs in this study are finite, simple (hence without loops or multiple edges), and connected – unless stated otherwise. Furthermore we make the standard assumption that graphs are encoded as adjacency lists. Given a graph $G = (V,E)$ and a vertex $v \in V$, we denote its neighbourhood by $N_G(v) = \{u \in V \mid \{u,v\} \in E\}$ and the set of its incident edges by $E_v(G) = \{\{u,v\} \mid u \in N_G(v)\}$. When $G$ is clear from the context we write $N(v)$ and $E_v$ instead of $N_G(v)$ and $E_v(G)$. Similarly, we define the neighbourhood of any vertex-subset $S \subseteq V$ as $N_G(S) = (\bigcup_{v \in S} N_G(v)) \setminus S$.

2.1 Split-width

Let a split in a graph $G = (V,E)$ be a partition $V = U \cup W$ such that: $\min\{|U|,|W|\} \geq 2$; and there is a complete join between the vertices of $N_G(U)$ and $N_G(W)$. A simple decomposition of $G$ takes as input a split $(U,W)$, and it outputs two subgraphs $G_U = G[U \cup \{w\}]$ and $G_W = G[W \cup \{u\}]$ where $u,w \notin V$ are fresh new vertices such that $N_{G_U}(w) = U$ and $N_{G_W}(u) = W$. The vertices $u,w$ are termed split marker vertices. A split decomposition of $G$ is obtained by applying recursively some sequence of simple decompositions (e.g., see Fig. 1). We name split components the subgraphs in a given split decomposition of $G$.

It is often desirable to apply simple decompositions until all the subgraphs obtained cannot be further decomposed. In the literature there are two cases of “indecomposable” graphs. Degenerate graphs are such that every bipartition of their vertex-set is a split. They are exactly the complete graphs and the stars [6]. A graph is prime for split decomposition if it has no split. We can define the following two types of split decomposition:

- **Canonical split decomposition.** Every graph has a canonical split decomposition where all the subgraphs obtained are either degenerate or prime and the number of subgraphs is minimized. Furthermore, the canonical split decomposition of a given graph can be computed in linear-time [4].

- **Minimal split decomposition.** A split-decomposition is minimal if all the subgraphs obtained are prime. A minimal split-decomposition can be computed from the canonical split-decomposition in linear-time [6]. Doing so, we avoid handling with the particular cases of stars and complete graphs in our algorithms. The set of prime graphs in any minimal split decomposition is unique up to isomorphism [6].
For instance, the split decomposition of Fig. 1 is both minimal and canonical.

Definition 1. The split-width of $G$, denoted by $sw(G)$, is the minimum $k \geq 2$ such that any prime subgraph in the canonical split decomposition of $G$ has order at most $k$.

We refer to [23] for some algorithmic applications of split decomposition. In particular, graphs with split-width at most two are exactly the distance-hereditary graphs, a.k.a the graphs whose all connected induced subgraphs are distance-preserving [1]. Distance-hereditary graphs contain many interesting subclasses of their own such as cographs (a.k.a., $P_4$-free graphs) and 3-leaf powers. Furthermore, since every degenerate graph has a split decomposition where all the components are either triangles or paths of length three, every component in a minimal split decomposition of $G$ has order at most $\max\{3, sw(G)\}$.

Split decomposition tree. A split decomposition tree of $G$ is a tree $T$ where the nodes are in bijective correspondence with the subgraphs of a given split decomposition of $G$, and the edges of $T$ are in bijective correspondence with the simple decompositions used for their computation. More precisely, if the considered split decomposition is reduced to $G$ then $T$ is reduced to a single node; Otherwise, let $(U, W)$ be a split of $G$ and let $G_U = (U \cup \{w\}, E_U), \ G_W = (W \cup \{u\}, E_W)$ be the corresponding subgraphs of $G$. We construct the split decomposition trees $T_U, T_W$ for $G_U$ and $G_W$, respectively. Furthermore, the split marker vertices $u$ and $w$ are contained in a unique split component of $G_W$ and $G_U$, respectively. We obtain $T$ from $T_U$ and $T_W$ by adding an edge between the two nodes that correspond to these subgraphs. The split decomposition tree of the canonical split decomposition, resp. of a minimal split decomposition, can be constructed in linear-time [23].

2.2 Matching problems

A matching in a graph is a set of edges with pairwise disjoint end vertices.

Problem 2 (Maximum-Cardinality Matching).

Input: A graph $G = (V, E)$.

Output: A matching of $G$ with maximum cardinality.

The Maximum-Cardinality Matching problem can be solved in $O(m\sqrt{n})$-time [22]. We do not use this result directly in our paper. However, we do use in our analysis the notion of augmenting paths, that is a cornerstone of most matching algorithms. Namely, let $G = (V, E)$ be a graph and $F \subseteq E$ be a matching of $G$. A vertex is termed matched if it is incident to an edge of $F$, and exposed otherwise. An $F$-augmenting path is a path where the two ends are exposed, all edges $\{v_{2i}, v_{2i+1}\}$ are in $F$ and all edges $\{v_{2j-1}, v_{2j}\}$ are not in $F$. We can observe that, given an $F$-augmenting path $P = (v_1, v_2, \ldots, v_{2\ell})$, the matching $E(P) \Delta F$ (obtained by replacing the edges $\{v_{2i}, v_{2i+1}\}$ with the edges $\{v_{2j-1}, v_{2j}\}$) has larger cardinality than $F$.

Lemma 3 (Berge, [2]). A matching $F$ in $G = (V, E)$ is maximum if and only if there is no $F$-augmenting path.

It is folklore that the proof of Berge’s lemma also implies the existence of many vertex-disjoint augmenting paths for small matchings. More precisely:

Lemma 4 (Hopcroft-Karp, [18]). Let $F_1, F_2$ be matchings in $G = (V, E)$. If $|F_1| = r$, $|F_2| = s$ and $s > r$, then there exist at least $s - r$ vertex-disjoint $F_1$-augmenting paths.
**b-Matching.** More generally given a graph $G = (V, E)$, let $b : V \rightarrow \mathbb{N}$ assign a nonnegative integer capacity $b_v$ for every vertex $v \in V$. A $b$-matching is an assignment of nonnegative integer edge-weights $(x_e)_{e \in E}$ such that, for every $v \in V$, we have $\sum_{e \in E_v} x_e \leq b_v$. We define the $x$-degree of vertex $v$ as $\deg_x(v) = \sum_{e \in E_v} x_e$. Furthermore, the cardinality of a $b$-matching is defined as $||x||_1 = \sum_{e \in E} x_e$. We will consider the following graph problem:

**Problem 5 (b-Matching).**

**Input:** A graph $G = (V, E)$; an assignment function $b : V \rightarrow \mathbb{N}$.

**Output:** A $b$-matching of $G$ with maximum cardinality.

For technical reasons, we will also use the following variant of $b$-Matching. Let $c : E \rightarrow \mathbb{N}$ assign a cost to every edge. The cost of a given $b$-matching $x$ is defined as $c \cdot x = \sum_{e \in E} c_e x_e$.

**Problem 6 (Maximum-Cost b-Matching).**

**Input:** A graph $G = (V, E)$; assignment functions $b : V \rightarrow \mathbb{N}$ and $c : E \rightarrow \mathbb{N}$.

**Output:** A maximum-cardinality $b$-matching of $G$ where the cost is maximized.

**Lemma 7 ( [14, 15]).** For every $G = (V, E)$ and $b : V \rightarrow \mathbb{N}, c : E \rightarrow \mathbb{N}$, we can solve Maximum-Cost $b$-Matching in $O(nm \log^2 n)$-time.

In particular, we can solve $b$-Matching in $O(nm \log^2 n)$-time.

There is a nonefficient (quasi polynomial) reduction from $b$-Matching to Maximum-Cardinality Matching that we will use in our analysis (e.g., see [25]). More precisely, let $G, b$ be any instance of $b$-Matching. The “expanded graph” $G_b$ is obtained from $G$ and $b$ as follows. For every $v \in V$, we add the nonadjacent vertices $v_1, v_2, \ldots, v_{b_v}$ in $G_b$. Then, for every $\{u, v\} \in E$, we add the edges $\{u_i, v_j\}$ in $G_b$, for every $1 \leq i \leq b_u$ and for every $1 \leq j \leq b_v$. It is easy to transform any $b$-matching of $G$ into an ordinary matching of $G_b$, and vice-versa.

### 2.3 High-level presentation of the algorithm

In order to discuss the difficulties we had to face on, we start giving an overview of the FPT algorithms that are based on split decomposition.

- We first need to define a vertex-weighted variant of the problem that needs to be solved for every component of the decomposition separately (possibly more than once). This is because there are split marker vertices in every component that substitute the other remaining components; intuitively, the weight of such a vertex encodes a partial solution for the union of split components it has substituted.

- Then, we take advantage of the treelike structure of split decomposition in order to solve the weighted problem, for every split component sequentially, using dynamic programming. Roughly, this part of the algorithm is based on a split decomposition tree. Starting from the leaves of that tree (resp., from the root), we perform a tree traversal. For every split component, we can precompute its vertex-weights from the partial solutions we obtained for its children (resp., for its father) in the split decomposition tree.

**Our approach.** In our case, a natural vertex-weighted generalization for Maximum-Cardinality Matching is the unit-cost $b$-Matching problem [12]. Independently from this work\(^1\), the authors in [20] proposed a new Maximum-Cardinality Matching algorithm.

---

\(^1\) Our preliminary version of this paper was released on arXiv one day before theirs.
on graphs of bounded modular-width that is also based on a reduction to $b$-Matching. Unlike this work, the algorithm of [20] cannot be applied to the more general case of bounded split-width graphs. Indeed, the main technical difficulty for the latter graphs – not addressed in [20] – is how to precompute efficiently, for every component of their split decomposition, the specific instances of $b$-Matching that need to be solved. To see that, consider the bipartition $(U, W)$ that results from the removal of a split. In order to compute the $b$-Matching instances on side $U$, we should be able (after processing the other side $W$) to determine the number of edges of the split that are matched in a final solution. Guessing such number looks computationally challenging. We avoid doing so by storing a partial solution for every possible number of split edges that can be matched. However, this simple approach suffers from several limitations. For instance, we need a very compact encoding for partial solutions – otherwise we could not achieve a quasi linear-time complexity. Somehow, we also need to consider the partial solutions for all the splits that are incident to the same component all at once.

This is where we use a result from Section 3, namely, that for every fixed vertex $w$ in a graph, the maximum cardinality of a $b$-matching is a piecewise-linear function in the capacity $b_w$ of this vertex. Roughly, in any given split component $C_i$, we consider all the vertices $w$ substituting a union of other components. The latter vertices are in one-to-one correspondence with the strong splits that are incident to the component. We expand every such vertex $w$ to a module that contains $O(1)$ vertices for every straight-line section of the corresponding piecewise-linear function. We want to stress that to the best of our knowledge, the combination of dynamic programming over split decomposition with the recursive computation of some piecewise-linear functions is an all new algorithmic technique.

### 3 Changing the capacity of one vertex

We first consider an auxiliary problem on $b$-matching that can be of independent interest. Let $G = (V, E)$ be a graph, $w \in V$ and $b : V \setminus w \to \mathbb{N}$ be a partial assignment. We denote $\mu(t)$ the maximum cardinality of a $b$-matching of $G$ provided we set to $t$ the capacity of vertex $w$. Clearly, $\mu$ is nondecreasing in $t$. Our main result in this section is that the function $\mu$ is essentially piecewise linear (Proposition 11). We start by introducing some useful lemmata.

- **Lemma 8.** $\mu(t + 1) - \mu(t) \leq 1$.
- **Lemma 9.** If $\mu(t + 2) = \mu(t)$ then we have $\mu(t + i) = \mu(t)$ for every $i \geq 0$.
- **Lemma 10.** If $\mu(t + 1) = \mu(t)$ then we have $\mu(t + 3) = \mu(t + 2)$.

These above results are obtained by studying vertex-disjoint augmenting paths in some “expanded graphs” $G_{b,t}$ (cf. Lemmata 3 and 4).

- **Proposition 11.** There exist integers $c_1, c_2$ such that:

$$
\mu(t) = \begin{cases} 
\mu(0) + t & \text{if } t \leq c_1 \\
\mu(c_1) + \left\lfloor \frac{t - c_1}{2} \right\rfloor = \mu(0) + c_1 + \left\lfloor \frac{t - c_1}{2} \right\rfloor & \text{if } c_1 < t \leq c_1 + 2c_2 \\
\mu(c_1 + 2c_2) = \mu(0) + c_1 + c_2 & \text{otherwise.}
\end{cases}
$$

Furthermore, the triple $(\mu(0), c_1, c_2)$ can be computed in $O(nm \log^2 n \log |b|_1)$-time.
The b-Matching Problem in Distance-Hereditary Graphs and Beyond

We root the split decomposition tree T of any minimal split decomposition of G. We root T in an arbitrary component C1. Then, starting from the leaves, we compute by dynamic programming on T the cardinality of an optimal solution. This first part of the algorithm is involved, and it uses the results of Section 3. It is based on a new reduction rule that we introduce in Definition 12. Finally, starting from the root component C1, we compute a maximum-cardinality b-matching of G, b by reverse dynamic programming on T. This second part of the algorithm is simpler than the first one, but we need to carefully upper-bound its time complexity. In particular, we also need to ensure that some additional property holds for the b-matchings we compute at every component.

Proof. Let c1 be the maximum integer t such that μ(t) = μ(0) + t. This value is well-defined since μ must stay constant whenever t ≥ ∑w∈N_G(w) b_w (saturation of all the neighbours). Furthermore, by Lemma 8 we have μ(t) = μ(0) + t for every 0 ≤ t ≤ c1. Then, let t_max be the least integer t such that, for every i ≥ 0 we have μ(t_max + i) = μ(t_max). Again, this value is well-defined since we have the trivial upper-bound t_max ≤ ∑w∈N_G(w) b_w. Furthermore, since μ is strictly increasing between 0 and c1, t_max ≥ c1. Let c_2 = t_max − c1. We claim that c_2 = 2c_2 is even. For that, we need to observe that μ(c_1) = μ(c_1 + 1) by maximality of c_1. Using Lemma 10, we prove by induction μ(c_1 + 2i) = μ(c_1 + 2i + 1) for every i ≥ 0. The latter proves, as claimed, c_2 = 2c_2 is even by minimality of c_2. Moreover, for every 0 ≤ i ≤ c_3 we have by Lemma 9 μ(c_1 + 2i) = μ(c_1 + 2i + 1) (since otherwise t_max ≤ c_1 + 2i). By Lemma 10 we have μ(c_1 + 2i) = μ(c_1 + 2i + 1). Finally, by Lemma 8 we get μ(c_1 + 2(i + 1)) ≤ μ(c_1 + 2i + 1) + 1 = μ(c_1 + 2i) + 1, therefore μ(c_1 + 2(i + 1)) = μ(c_1 + 2i) + 1. Altogether combined, it implies that μ(c_1 + 2i) = μ(c_1 + 2i + 1) = μ(c_1 + i) + 1 for every 0 ≤ i ≤ c_2, that proves the first part of our result.

We can compute μ(0) with any b-MATCHING algorithm after we set the capacity of w to 0. The value of c_1 can be computed within O(log c_1) calls to a b-MATCHING algorithm, as follows. Starting from c_1 = 1, we multiply the current value of c_1 by 2 until we reach a value c_i such that μ c_i < μ(0) + c_1. Then, we perform a binary search between 0 and c_i in order to find the largest value c_1 such that μ(c_1) = μ(0) + 1. Once c_1 is known, we can use a similar approach in order to compute c_2. Overall, since c_1 + 2c_2 = t_max ≤ ∑w∈N_G(w) b_w = O(||b||_1), we are left with O(log ||b||_1) calls to any b-MATCHING algorithm. Therefore, by Lemma 7, we can compute the triple (μ(0), c_1, c_2) in O(nm log^2 n log ||b||_1)-time.

4 The algorithm

We present in this section a quasi linear-time algorithm for computing a maximum-cardinality b-matching on any bounded split-width graph (Theorem 17). Given a graph G, our algorithm takes as input the split decomposition split-width graph of G. We root T in an arbitrary component C1. Then, starting from the leaves, we compute by dynamic programming on T the cardinality of an optimal solution. This first part of the algorithm is involved, and it uses the results of Section 3. It is based on a new reduction rule that we introduce in Definition 12. Finally, starting from the root component C1, we compute a maximum-cardinality b-matching of G, b by reverse dynamic programming on T. This second part of the algorithm is simpler than the first one, but we need to carefully upper-bound its time complexity. In particular, we also need to ensure that some additional property holds for the b-matchings we compute at every component.
4.1 Reduction rule

Recall that an edge between a rooted subtree and its parent in $T$ corresponds to a split $(U,W)$ of $G$. After we processed the side $U$ (corresponding to this subtree) we account for all the partial solutions found for $G_U$ by transforming the split marker vertex $u$ into a module $^\text{2}$, as follows:

**Definition 12.** For any instance $G = (V,E)$, $b$ and any split $(U,W)$ of $G$ let $C = N_G(W) \subseteq U$, $D = N_G(U) \subseteq W$. Let $G_U = (U \cup \{w\}, E_U)$, $G_W = (W \cup \{u\}, E_W)$ be the corresponding subgraphs of $G$. We define the pairs $G_U, b_U$ and $H_W, b_W$ as follows:

- For every $v \in U$ we set $b_U^v = b_v$; the capacity of the split marker vertex $w$ is left unspecified.
- Let $(\mu_U^U(0), c_U^1, c_U^2)$ be as defined in Proposition 11 w.r.t. $G_U, b_U$ and $w$.
- The auxiliary graph $H_W$ is obtained from $G_W$ by replacing the split marker vertex $u$ by a module $M_u = \{u_1, u_2, u_3\}$, $N_{H_W}(M_u) = N_{G_W}(u) = D$: we also add an edge between $u_2, u_3$. For every $v \in W$ we set $b_W^v = b_v$; we set $b_W^{u_1} = c_1^U$, $b_W^{u_2} = b_W^{u_3} = c_2^U$.

See Fig. 3 for an illustration. We will show throughout this section that our gadget somewhat encodes all the partial solutions for side $U$. Formally, the following relationship holds between solutions for $G, b$ and solutions for $H_W, b_W$:

**Proposition 13.** Given a graph $G = (V,E)$ and a capacity function $b$, let $(U,W)$ be a split of $G$ and let $H_W, b_W$ be as in Definition 12. If $x$ and $x^W$ are maximum-cardinality $b$-matchings for the pairs $G, b$ and $H_W, b_W$, respectively, then we have:

$$ ||x||_1 = ||x^W||_1 + \mu_U^U(0) - c_2^U $$

In what follows, we prove the first direction of Proposition 13 using classical flow techniques. We postpone the proof of the other direction since, for that one, we need to prove intermediate lemmata that will be also used in the proof of Theorem 17.

**Lemma 14.** Let $x$ be a $b$-matching for $G, b$. There exists a $b$-matching $x^W$ for $H_W, b_W$ such that $||x^W||_1 \geq ||x||_1 + c_2^U - \mu_U^U(0)$.

The following Sections 4.2 and 4.3 detail the intermediate results that we will use in order to prove the other direction of Proposition 13 (as well as Theorem 17).

4.2 $b$-matchings with additional properties

We consider an intermediate modification problem on the $b$-matchings of some “auxiliary graphs” that we define next. Let $C_i$ be a split component in a given split decomposition of $G$. The subgraph $C_i$ is obtained from a sequence of simple decompositions. For a given subsequence of the above simple decompositions (corresponding to the edges between $C_i$ and

---

2 Recall that $M$ is a module if for every $x,y \in M$ we have $N(x) \setminus M = N(y) \setminus M$. 

---

ISAAC 2018
its children in $T$) we apply the reduction rule of Definition 12. Doing so, we obtain a pair $H_i, b'$ with $H_i$ being a supergraph of $C_i$ obtained by replacing some split marker vertices $u_{i_t}, 1 \leq t \leq \ell$, by the modules $M_{i_t} = \{u_{i_t}^1, u_{i_t}^2, u_{i_t}^3\}$. By construction $u_{i_t}^2, u_{i_t}^3$ are adjacent and they have the same capacity.

We seek for a maximum-cardinality $b$-matching $x^i$ for the pair $H_i, b'$ such that the following properties hold for every $1 \leq t \leq \ell$:

- (symmetry) $\deg_{x^i}(u_{i_t}^2) = \deg_{x^i}(u_{i_t}^3)$.
- (saturation) if $\deg_{x^i}(u_{i_t}^1) < c_{i_t}^1$ then, $\deg_{x^i}(u_{i_t}^2) = x^i_{\{u_{i_t}^3, u_{i_t}^2\}}$.

We prove next that for every fixed $t$, any $x^i$ can be processed in $O(|E_{u_{i_t}}(C_i)|)$-time so that both the saturation property and the symmetry property hold for $M_{i_t}$. However, ensuring that these two above properties hold simultaneously for every $t$ happens to be trickier. We manage to do so by reducing to MAXIMUM-COST $b$-MATCHING (i.e., internal edges in the modules are assigned a larger cost than the other edges).

Lemma 15. In $O(|V(H_i)| \cdot |E(H_i)| \cdot \log^2 |V(H_i)|)$-time, we can compute a maximum-cardinality $b$-matching $x^i$ for the pair $H_i, b'$ such that both the saturation property and the symmetry property hold for every $M_{i_t}, 1 \leq t \leq \ell$.

4.3 Merging the partial solutions together

Finally, before we can describe our main algorithm (Theorem 17) we need to consider the intermediate problem of merging two partial solutions. Let $(U, W)$ be a split of $G$ and let $G_U = (U \cup \{w\}, E_U)$, $G_W = (W \cup \{u\}, E_W)$ be the corresponding subgraphs of $G$. Consider some partial solutions $x^U$ and $x^W$ obtained, respectively, for the pairs $G_U, b^U$ and $G_W, b^W$ (for some $b^U, b^W$ to be defined later). Assuming an appropriate data-structure for $b$-matchings, this merging stage can be solved with a greedy algorithm.

Lemma 16. Suppose $b^U$ (resp., $b^W$) satisfies $b^U_v \leq b_v$ for every $v \in U$ (resp., $b^W_u \leq b_u$ for every $v \in W$). Let $x^U, x^W$ be $b$-matchings for, respectively, the pairs $G_U, b^U$ and $G_W, b^W$ such that $\deg_{x^U}(w) = \deg_{x^W}(u) = d$.

Furthermore, for any graph $H$ let $\varphi(H) = |E(H)| + 4 \cdot (\text{sc}(H) - 1)$, with $\text{sc}(H)$ being the number of split components in any minimal split decomposition of $H$.\footnote{We recall that the set of prime graphs in any minimal split decomposition is unique up to isomorphism [23].}

Then, in at most $O(\varphi(G) - \varphi(G_U) - \varphi(G_W))$-time, we can obtain a valid $b$-matching $x$ for the pair $G, b$ such that $||x||_1 = ||x^U||_1 + ||x^W||_1 - d$.

Overall, since there are at most $n - 2$ components in any minimal split decomposition of $G$ [23], the merging stages take total time $O(\varphi(G)) = O(n + m)$.

4.4 Main result

We are now ready to prove Proposition 13. This algorithmic proof is the cornerstone of our main result.

Proof of Proposition 13. We have $||x^W||_1 \geq ||x||_1 - \mu^U(0) + c_1^1$ by Lemma 14. In order to prove the converse inequality, we can assume w.l.o.g. that $x^W$ satisfies both the saturation property and the symmetry property w.r.t. the module $M_u$ (otherwise, by Lemma 15, we can process $x^W$ so that it is the case). We partition $||x^W||_1$ as follows: $\mu^W = \sum_{e \in E(W)} x^W_e$, $c_1^1 = \deg_{x^W}(u_1) \leq c^1_1$ and $c_2 = \deg_{x^W}(u_2) - x^W_{\{u_2, u_3\}} = \deg_{x^W}(u_3) - x^W_{\{u_4, u_5\}} \leq c^2_2$. Since we
assume that $x^W$ satisfies both the saturation property and the symmetry property w.r.t. $M_u$, we have $c_2' > 0$ only if $c_1' = c_2'$. Furthermore, we observe that $u_2$ and $u_3$ must be saturated (otherwise, we could increase the cardinality of the $b$-matching by setting $x_{\{u_2,u_3\}}^W = c_2' - c_2$).

Therefore, we get:

$$||x^W||_1 = \mu^W + c_1' + 2c_2' + (c_2' - c_2) = \mu^W + c_1' + c_2' - c_2.$$

We define $b_u^W = b_u'^W = c_1' + 2c_2'$. Then, we proceed as follows (see Fig. 4 for an illustration).

- We transform $x^W$ into a $b$-matching for the pair $G_{W}, b^W$ by setting $x_{\{u,v\}}^W = x_{\{u_1,v\}}^W + x_{\{u_2,v\}}^W + x_{\{u_3,v\}}^W$ for every $v' \in N_{G_W}(u) = D$. Note that we have $\delta_{G_{W}}(u) = b_u^W = c_1' + 2c_2'$. Furthermore, the cardinality of the $b$-matching has decreased by $x_{\{u_2,u_3\}}^W = c_2' - c_2$.

- Let $x^U$ be a $b$-matching for the pair $G_{U}, b^U$ of maximum cardinality $\mu^U(c_1' + 2c_2')$. Since $c_1' \leq c_1', c_2' > 0$ only if $c_1' = c_2'$, and $c_2' \leq c_2'$, the following can be deduced from Proposition 11: $||x^U||_1 = \mu^U(c_1' + 2c_2') = \mu^U(0) + c_1' + c_2'$ and the split marker vertex $w$ is saturated in $x^U$, i.e., $\delta_{G_{U}}(w) = b_w^U = c_1' + 2c_2'$.

Since we have $\delta_{G_{U}}(w) = \delta_{G_{W}}(w) = c_1' + 2c_2'$, we can define a $b$-matching $x'$ for the pair $G, b$ by applying Lemma 16. Doing so, we get $||x||_1 \geq ||x'||_1 = ||x^U||_1 + (||x^W||_1 - (c_2' - c_2)) - (c_1' + 2c_2') = \mu^U(0) + c_1' + c_2' + ||x^W||_1 - (c_2' + c_2') = ||x^W||_1 + \mu^U(0) - c_2'.

We finally prove (in a similar way as above) the main result in this paper.

**Theorem 17.** For every pair $G = (V, E), b$ with $sw(G) \leq k$, we can solve $b$-Matching in $O((k \log k) \cdot (m + n) \cdot \log (||b||_1))$-time.

Setting $b_v = 1$ for every $v \in V$, we obtain the following implication of Theorem 17:

**Corollary 18.** For every graph $G = (V, E)$ with $sw(G) \leq k$, we can solve Maximum-Cardinality Matching in $O((k \log k) \cdot (m + n) \cdot \log n)$-time.

### 5 Open questions

We presented an algorithm for solving $b$-Matching on distance-hereditary graphs, and more generally on any graph with bounded split-width. In contrast to our result, we stress that as already noticed in [20], Maximum-Weight Matching cannot be solved faster on complete graphs, and so, on distance-hereditary graphs, than on general graphs. An interesting open question would be to know whether $b$-Matching can be solved in linear time on bounded split-width graphs. In a companion paper [10], we prove with a completely different approach that Maximum-Cardinality Matching can be solved in $O(n + m)$-time on distance-hereditary graphs. However, it is not clear to us whether similar techniques can be used for bounded split-width graphs in general.

New Algorithms for Edge Induced König-Egerváry Subgraph Based on Gallai-Edmonds Decomposition

Qilong Feng
School of Information Science and Engineering, Central South University, Changsha, P.R. China
csuufeng@mail.csu.edu.cn

Guanlan Tan
School of Information Science and Engineering, Central South University, Changsha, P.R. China

Senmin Zhu
School of Information Science and Engineering, Central South University, Changsha, P.R. China

Bin Fu
Department of Computer Science, University of Texas-Rio Grande Valley, USA

Jianxin Wang
School of Information Science and Engineering, Central South University, Changsha, P.R. China

Abstract

König-Egerváry graphs form an important graph class which has been studied extensively in graph theory. Much attention has also been paid on König-Egerváry subgraphs and König-Egerváry graph modification problems. In this paper, we focus on one König-Egerváry subgraph problem, called the Maximum Edge Induced König Subgraph problem. By exploiting the classical Gallai-Edmonds decomposition, we establish connections between minimum vertex cover, Gallai-Edmonds decomposition structure, maximum matching, maximum bisection, and König-Egerváry subgraph structure. We obtain a new structural property of König-Egerváry subgraph: every graph \( G = (V, E) \) has an edge induced König-Egerváry subgraph with at least \( 2|E|/3 \) edges. Based on the new structural property proposed, an approximation algorithm with ratio \( 10/7 \) for the Maximum Edge Induced König Subgraph problem is presented, improving the current best ratio of \( 5/3 \). To the best of our knowledge, this paper is the first one establishing the connection between Gallai-Edmonds decomposition and König-Egerváry graphs. Using \( 2|E|/3 \) as a lower bound, we define the Edge Induced König Subgraph above lower bound problem, and give a kernel of at most \( 30k \) edges for the problem.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms, Mathematics of computing → Approximation algorithms

Keywords and phrases König-Egerváry graph, Gallai-Edmonds decomposition

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.31

Funding This work is supported by the National Natural Science Foundation of China under Grants (61420106009, 61872450, 61828205, 61672536).
Introduction

Given a graph $G$, a matching $M$ in $G$ is a set of vertex-disjoint edges. Matching problem is one of the fundamental problems in combinatorial optimization, and has wide applications in many fields. For several decades, much attention has been paid on matching and related problems.

The Vertex Cover problem is closely related to the matching problem, which is to decide, for a given graph $G = (V, E)$, whether there exists a subset $V' \subseteq V$ of at most $k$ vertices such that each edge in $G$ has at least one endpoint in $V'$. The Vertex Cover problem is one of the 21 NP-complete problems [19], and has been extensively studied in the field of parameterized complexity [8, 15, 22, 32, 36]. The current best parameterized algorithm for the Vertex Cover problem is of running time $O^*(1.2738^k)$ [8], where $k$ is the size of vertex cover in given graph. Matching methods can also be applied to deal with the Vertex Cover problem. For the bipartite graphs, it is proved that the size of a minimum vertex cover is equal to the size of a maximum matching [1]. Thus, the Vertex Cover problem on bipartite graphs can be solved in polynomial time based on the algorithms of getting a maximum matching. For general graphs, based on the maximum matching, an approximation algorithm with ratio 2 can be obtained for the Vertex Cover problem, which is still the current best approximation ratio for the problem. By using matching number as a lower bound, a variant of the Vertex Cover problem, called Above-Guarantee Vertex Cover problem (given a graph $G$ and parameter $k$, decide whether $G$ has a vertex cover of size at most $|M| + k$, where $M$ is a maximum matching in $G$) was first studied in [40]. Thereafter, several interesting results for the Above-Guarantee Vertex Cover problem have been obtained [9, 15, 25, 36, 37, 34].

The classical Gallai-Edmonds decomposition method provides an elegant structure for graphs based on matching. For any graph $G$, a Gallai-Edmonds decomposition of graph $G$ can be obtained in polynomial time [27], which is a tuple $(X, Y, Z)$, where $X$ is the set of vertices in $G$ which are not covered by at least one maximum matching of $G$, $Y$ is $N(X)$ ($N(X)$ is the set of neighbors of the vertices in $X$ with $N(X) \cap X = \emptyset$), and $Z = V(G) \setminus (X \cup Y)$. The application of Gallai-Edmonds decomposition has been paid lots of attention, and many problems were studied by applying Gallai-Edmonds decomposition from approximation algorithms and parameterized complexity points of view, such as approximation algorithms [14, 28, 35], kernelizations [13, 21, 33], parameterized algorithms [7, 11, 15], etc. Gallai-Edmonds decomposition has also been applied to solve problems in many other fields [2, 3, 18, 38].

A graph $G$ is a König-Egerváry graph (in short, König graph) if the size of a minimum vertex cover of $G$ is equal to the size of a maximum matching of $G$. The structural properties of König graphs have been studied for a long time. Deming [10] studied the characterizations of König graphs through independence number of graphs, and proved that the König graphs can be recognized in polynomial time. Stersoul [39] studied the characterizations of König graphs through the structure of matchings in graphs. Lovász [26] studied König graphs with perfect matching, and gave the excluded subgraphs characterizations through matching-covered graphs. Bourjolly and Pulleyblank [5] studied the relation between König graphs and 2-bicritical graphs, and showed that the characterizations of König graphs can be used to obtain a structural characterization of 2-bicritical graphs. Korach, Nguyen, and Peis [20] studied subgraph characterizations of Red/Blue-Split graphs and König graphs, where Red/Blue-Split graphs are the generalizations of König graphs and Split graphs. Levit and Mandrescu [23] studied the relation between critical independent sets and König graphs.

In this paper, we focus on the König-Egerváry subgraph problem, and study the problem from approximation algorithm and parameterized complexity points of view. For a graph \(G = (V, E)\) and a subset \(E' \subseteq E\), the subgraph induced by edges in \(E'\), denoted by \(G[E']\), is the one that contains the endpoints of the edges in \(E'\), and contains the edges in \(E'\). If the size of a minimum vertex cover is equal to the size of a maximum matching in \(G[E']\), then \(G[E']\) is called a König subgraph. We now give the definitions of the related problems.

**Maximum Edge Induced König Subgraph:**
Given a graph \(G = (V, E)\), find a set \(E' \subseteq E\) with maximum number of edges such that the edges in \(E'\) induce a König subgraph.

**Edge Induced König Subgraph:**
Given a graph \(G = (V, E)\) and non-negative integer \(k\), find a set \(E'\) of at least \(k\) edges in \(E\) such that the edges in \(E'\) induce a König subgraph, or report that no such set exists.

The Edge Induced König Subgraph problem is closely related to a graph modification problem, called König Edge Deletion problem, which is to delete at most \(k\) edges to turn a given graph into a König graph. For the NP-completeness, the Edge Induced König Subgraph problem and the König Edge Deletion problem are equivalent. However, the approximability and parameterized complexity of those two problems are totally different. For the Edge Induced König Subgraph problem, Mishra et al. [32] presented an approximation algorithm with ratio \(5/3\), and gave a parameterized algorithm of running time \(O^*(2^k)\). For the König Edge Deletion problem, Mishra et al. [32] proved that this problem does not admit any constant-factor approximation algorithm unless UGC fails. As pointed out in [30, 31, 32], the parameterized complexity of the König Edge Deletion problem is still open. On the other hand, many other König subgraph and König graph problems have also been studied. Mishra et al. [30, 32] studied the Vertex Induced König Subgraph problem (given a graph \(G\) and non-negative integer \(k\), decide whether there exists a set of at least \(k\) vertices that induces a König subgraph) and the König Vertex Deletion problem (given a graph \(G\) and non-negative integer \(k\), decide whether there exists a set of at most \(k\) vertices whose deletion results in a König subgraph). For the Vertex Induced König Subgraph problem, Mishra et al. [32] proved that it is \(W[1]\)-hard. For the König Vertex Deletion problem, a series of parameterized algorithms have been proposed [9, 25, 30, 32]. As the generalizations of König graphs and Split graphs, Red/Blue-Split graph modification problems have also been studied [20, 29, 30].

In this paper, we study the Edge Induced König Subgraph problem from approximation and parameterized algorithms points of view. The main contribution of this paper is that we present structural connections between minimum vertex cover, Gallai-Edmonds decomposition, maximum bisection, and König subgraphs, get a new structural property for the König subgraph of a given graph, and propose an improved approximation algorithm for the Edge Induced König Subgraph problem. To the best of our knowledge, this paper is the first one to establish connection between Gallai-Edmonds decomposition and the structures of König graphs.
We now point out the differences of our techniques and results in this paper with the ones in [31, 32].

(1) The 5/3-approximation algorithm for the Maximum Edge Induced König Subgraph problem in [31, 32] is based on an important property of König subgraph: every graph $G$ has an edge induced König subgraph of at least $3m/5$ edges, where $m$ is the number of edges in $G$, which is obtained in [31, 32] based on the maximum matching in $G$. In this paper, we exploit the connection between Gallai-Edmonds decomposition and König subgraphs, and present a new structural property of König subgraphs: every $G$ has an edge induced König subgraph of at least $2m/3$ edges, which results in an improved approximation algorithm with ratio $10/7$.

(2) For a Gallai-Edmonds decomposition $(X,Y,Z)$ of given graph $G$, instead of directly applying the matching structure in the decomposition, we study the roles of factor-critical connected components of $G[X]$ to derive a König subgraph of $G$. For the connected components in $G[X]$, we use the “matching switching” strategy to analyze the number of edges from the connected components contained in the König subgraph, which is another key point to get the improved approximation algorithm for the problem.

(3) In this paper, we exploit a connection between structures of the König subgraphs and the properties of the Maximum Bisection above tight lower bound problem (given a graph $G = (V,E)$ and a parameter $k$, decide whether $V$ can be divided into two parts $V_1, V_2$ such that $|V_1| - |V_2| \leq 1$, and the number of edges with one endpoint in $V_1$ and the other endpoint in $V_2$ is at least $|E|/2 + k$). The kernelization results of the Maximum Bisection above tight lower bound problem are applied to analyze the size of the König subgraphs.

(4) For the parameterized algorithm of the Edge Induced König Subgraph problem, since we can get that every graph has an edge induced König subgraph of at least $2m/3$ edges, the parameter $k$ in the given instance of the Edge Induced König Subgraph problem is large. By using $2m/3$ as a lower bound, we propose a variant of the Edge Induced König Subgraph problem, called Edge Induced König Subgraph above lower bound problem, and give a kernel of at most $30k$ edges for the problem.

2 Preliminaries

Given a graph $G = (V,E)$, for two vertices $u,v$ in $G$, the edge between $u$ and $v$ if exists is denoted by $uv$. We say that edge $uv$ is incident to $u$ and $v$. For a vertex $v$ in $G$, the degree of $v$ denoted by $d(v)$ is the number of edges incident to $v$. For a subset $X \subseteq V$, $G[X]$ denotes the subgraph induced by the vertices in $X$. For a vertex $v$ in $X$, $d_X(v)$ denotes the degree of $v$ in the induced subgraph $G[X]$. For a matching $M$ in $G$, let $V(M)$ be the set of vertices contained in $M$. The vertices in $V(M)$ are the vertices matched by $M$, and it is also called that the vertices in $V(M)$ are saturated by $M$. The vertices in $V - V(M)$ are called unmatched vertices, and the edges in $M$ are called matched edges. A matching $M$ in $G$ is a perfect matching if all the vertices in $V$ are matched vertices. For a graph $G$ with $n$ vertices, if every (vertex) induced subgraph with $n - 1$ vertices has a perfect matching, then $G$ is called a factor-critical graph. For a matching $M$ in graph $G = (V,E)$, if $V(M)$ contains $|V| - 1$ vertices, then $M$ is called a near-perfect matching of $G$. A chord is an edge incident to two nonadjacent vertices in a cycle. A chordless cycle with at least four vertices is called a hole. For a subgraph $C$ in $G$, let $V(C)$ and $E(C)$ denote the sets of vertices and edges contained in $C$, respectively. For two subsets $A,B \subseteq V$, $E(A,B)$ is the set of edges, each of which has one endpoint in $A$ and the other endpoint in $B$. For a vertex $u$ and a subset $B \subseteq V$, for simplicity, let $E(u,B) = E(\{u\},B)$. For a partition $(V_1,V_2)$ of $V$, $(V_1,V_2)$ is
called a cut in $G$, and an edge with one endpoint in $V_1$ and the other endpoint in $V_2$ is called a cut edge of $(V_1, V_2)$. The size of cut $(V_1, V_2)$ is the number of cut edges in $E(V_1, V_2)$. A cut $(V_1, V_2)$ is called a bisection of $G$ if $|V_1| - |V_2| \leq 1$. A bisection with maximum number of cut edges is called a maximum bisection. A triangle is called a $C_3$.

- **Lemma 1** ([12, 27]). For a given graph $G$, the Gallai-Edmonds decomposition $(X, Y, Z)$ of $G$ has the following properties:
  1. the components of the subgraph induced by $X$ are factor-critical,
  2. the subgraph induced by $Z$ has a perfect matching,
  3. if $M$ is any maximum matching of $G$, it contains a near-perfect matching of each component of $G[X]$, a perfect matching of each component of $G[Z]$, and matches all vertices of $Y$ with vertices in distinct components of $G[X]$,
  4. the size of a maximum matching is $\frac{1}{2}(|V| - \delta(G[X]) + |Y|)$, where $\delta(G[X])$ is the number of connected components in $G[X]$.

3 New algorithms for Edge Induced König Subgraph

In this section, we give new structural properties of König subgraphs, and present an improved approximation algorithm for the Edge Induced König Subgraph problem. For a graph $G$, whether $G$ is a König graph or not can be decided by the following lemma.

- **Lemma 2** ([30, 31, 32]). A graph $G = (V, E)$ is a König graph if and only if there exists a cut $(V_1, V_2)$ of $V$ such that: (1) $V_1$ is a vertex cover of $G$; (2) there exists a matching across $(V_1, V_2)$ saturating each vertex in $V_1$.

We now give the relation between graphs with perfect matching and König graphs.

- **Lemma 3** ([31, 32]). Let $G = (V, E)$ be a graph with a perfect matching $M$, where $|V| = n$, $|E| = m$. Then a König subgraph $G'$ of $G$ with at least $3m/4 + n/8$ edges can be found in $O(mn)$ time such that $|M'| = |M|$, where $M'$ is a maximum matching in $G'$.

Given an instance $(G, k)$ of the Edge Induced König Subgraph problem, let $(X, Y, Z)$ be a Gallai-Edmonds decomposition of $G$. By Lemma 3, we get the following result.

- **Lemma 4.** Let $G_1$ be the subgraph induced by vertices in $Z$, and $M$ be a maximum matching in $G$. Then, there exists a König subgraph $G_1'$ in $G_1$ such that $|M'| = |E(G_1) \cap M|$, and $|E(G_1')| \geq 3|E(G_1)|/4$, where $M'$ is a maximum matching in $G_1'$.

Since each connected component $C$ of $G[X]$ is factor-critical, $C$ contains an odd number of vertices. Based on the degrees of the vertices in $X$ and a maximum matching $M$, we divide the vertices in $X$ into the following groups:

$X_1 = \{v \in X \mid d_X(v) = 0\}$,
$X_2 = \{v \in X \mid d_X(v) \geq 1, \exists u \in Y, uv \in M\}$,
$X_3 = \{v \in X \mid d_X(v) \geq 1, v \notin V(M)\}$.

Based on $X_1, X_2,$ and $X_3$, we divide the connected components of $G[X]$ into the following types.

1. $B_1$: each connected component of $B_1$ is an isolated vertex from $X_1$;
2. $B_2$: each connected component of $B_2$ contains a vertex from $X_2$;
3. $B_3$: each connected component of $B_3$ contains a vertex from $X_3$, and has exactly three vertices;
4. $B_5$: each connected component of $B_5$ contains a vertex from $X_3$, and has at least five vertices.
For each $B_i$ ($i = 1, 2, 3, 5$), let $V(B_i)$ and $E(B_i)$ be the sets of vertices and edges of $B_i$, respectively. For each connected component $C$ of $B_3$, let $a, b,$ and $c$ be the three vertices contained in $C$. By the definition of factor-critical, any two vertices from $\{a, b, c\}$ are adjacent. If $E(C, Y)$ is not empty, then arbitrarily choose any edge from $E(Y, C)$ (without loss of generality, assume that edge $ua$ is chosen). Then, edge $ua$ is called a *special edge*. Remark that any edge in $E(Y, C)$ can be viewed as special edge and only one edge from $E(Y, C)$ can be a special edge. For this case, if edge $bc$ is in maximum matching $M$, then $a$ is an unmatched vertex in $C$. We apply the strategy, called “matching switching”, to deal with the edges in $M \cap E(C)$, i.e., we delete $bc$ from $M$ and add edge $ab$ or $ac$ to $M$. It is easy to see that the new $M$ is still a maximum matching in $G$. After doing that, edge $bc$ is not an edge in $M$, which is called a *candidate deleted edge*. Let $SE$ be the set of special edges obtained by considering all connected components in $B_3$.

Given a graph $G$, we first give the relation between bisections and matchings in $G$.

**Lemma 5.** [16] Let $G$ be a graph and $M$ be a matching in $G$. Then $G$ has a bisection of size at least $\left\lceil m/2 \right\rceil + \left\lfloor |M|/2 \right\rfloor$, which can be found in $O(m + n)$ time, where $m, n$ are the number of edges and vertices in $G$, respectively.

For simplicity, we assume that all the numbers in the following are divisible, without any floor and ceiling notations.

We now analyze the relation between subgraph $G[Y \cup V(B_1) \cup V(B_2)]$ and König subgraphs.

**Lemma 6.** Let $G_2$ be the graph constructed by the subgraph $G[Y \cup V(B_1) \cup V(B_2)]$ and edges in $E(Y, Z)$, $E(Y, V(B_2)) \cup V(B_3) \setminus SE$. Then, there exists a König subgraph $G'_2$ in $G_2$ such that $|M'| = |M \cap E(G_2)| = |Y| + |M \cap E(B_2)|$, and $|E(G'_2)| \geq 11|E(G_2)|/15$, where $M'$ is a maximum matching in $G'_2$.

**Proof.** Assume that $B_2$ is not empty. Let $B_2 = \{b^2_1, \ldots, b^2_{h_2}\}$. For each connected component $b^2_i$ ($1 \leq i \leq h_2$), there must exist two vertices $u \in Y$ and $v \in V(b^2_i)$ such that edge $uv$ is in $M$. Add $u$ to a set $U$, which is initialized as an empty set. We need to consider the edges in $E(b^2_i)$, $E(u, Z)$, $E(u, X) \setminus SE$, and $E(u, Y)$. It is noted that for edges in $E(u, Y)$, there may exist another vertex $u'$ in $Y$ such that $E(u, Y) \cap E(u', Y) \neq \emptyset$. Therefore, in the process of analyzing the relation between $G[Y \cup V(B_1) \cup V(B_2)]$ and König subgraphs, we need to guarantee that each edge in $E(G[Y])$ can only be dealt with one time.

Since $b^2_i$ is factor-critical, subgraph $G[V(b^2_i) \setminus \{v\}]$ has a perfect matching, and the number of edges of $G[V(b^2_i) \setminus \{v\}]$ contained in $M$ is $(|V(b^2_i)| - 1)/2$. After dealing with all the connected components in $B_2$, $U$ contains $h_2$ vertices. For each vertex $u$ in $U$, there exists a connected component $b^2_i$ ($1 \leq i \leq h_2$) in $B_2$ and a vertex $v$ in $b^2_i$ such that $uv$ is in $M$. Let $Q^0_i = E(u, Z) \cup E(u, Y \setminus U) \cup E(u, X \setminus V(b^2_i)) \setminus SE$, $Q^1_i = E(v, V(b^2_i) \setminus \{v\})$, and $Q^2_i = E(G[V(b^2_i) \setminus \{v\}] ) \setminus M$.

Based on the analysis of the edges in $b^2_i$ and by Lemma 5, a bisection $(A_1, A_2)$ of size at least $m'/2 + (E(b^2_i) \cap M)/2$ in graph $G[V(b^2_i) \setminus \{v\}]$ can be found in $O(m' + |V(b^2_i) \setminus \{v\}|)$ time, where $m'$ is the number of edges in $G[V(b^2_i) \setminus \{v\}]$. Since $m' = |E(b^2_i) \cap M| + |Q^2_i|$, we get that the number of cut edges of bisection $(A_1, A_2)$ is at least $|E(b^2_i) \cap M| + |Q^2_i|/2$. It is easy to get that $|E(G[A_1])| + |E(G[A_2])| \leq |Q^2_i|/2$. Based on the sizes of $Q^0_i$ and $Q^1_i$, we now discuss how to delete edges to turn subgraph $G[V(b^2_i) \cup \{u\}]$ into a König subgraph.

**Case 1.** $|Q^0_i| \geq 3|Q^1_i|/8$. For this case, we put $u$ into the minimum vertex cover of $G$.

We will delete some edges in $Q^1_i$ and $Q^2_i$ to make $G[V(b^2_i) \cup \{u\}]$ be a König subgraph. We compare $|E(v, A_1)| + |E(G[A_1])|$ with $|E(v, A_2)| + |E(G[A_2])|$. Since $|E(v, A_1)| + |E(G[A_1])| + |E(v, A_2)| + |E(G[A_2])| \leq |Q^1_i| + |Q^2_i|/2$, one value of $|E(v, A_1)| + |E(G[A_1])| + |E(v, A_2)| + |E(G[A_2])|$
and $|E(v, A_2)| + |E(G[A_2])|$ is at most $(|Q_1^2| + |Q_2^2|)/2$. Without loss of generality, assume that $|E(v, A_2)| + |E(G[A_2])| \leq (|Q_1^2| + |Q_2^2|)/2$. We put the vertices in $A_1$ into the minimum vertex cover of $G$, and delete the edges in $E(v, A_2) \cup E(G[A_2])$ from subgraph $G[V(b_1^2) \cup \{u\}]$, and let $G'$ be the resulted subgraph. Since $uv \in M$ and $|M \cap E(G[V(b_1^2) \cup \{u\})| = (V(b_1^2) - 1)/2 + 1$, in the subgraph $G'$, the size of minimum vertex cover is $|A_1| + 1 = (V(b_1^2) - 1)/2 + 1$. Thus, subgraph $G'$ is a König subgraph. We now analyze the proportion of the deleted edges in $Q_1^0$ and $G[V(b_1^2) \cup \{u\}]$. Because vertex $u$ is contained in the minimum vertex cover, all the edges incident to $u$ are covered, i.e., the edges in $Q_2^0$ are covered by $u$. We get that
\begin{equation}
\frac{|E(v, A_2)| + |E(G[A_2])|}{|Q_1^0| + |Q_2^0| + |Q_1^2| + |M \cap E(b_1^2)| + 1} \leq \frac{|Q_1^0| + |Q_2^0| + |M \cap E(b_1^2)|}{|Q_1^0| + |Q_2^0| + |M \cap E(b_1^2)|} = 1
\end{equation}
Since $b_1^2$ is factor-critical, we have $|M \cap E(b_1^2)| \geq |Q_1^1|/2$. Therefore, for inequality 1, we get that
\begin{align}
\frac{|Q_1^0| + |Q_2^0|/2}{|Q_1^0| + |Q_2^0| + |Q_1^2| + |M \cap E(b_1^2)|/2} &\leq \frac{|Q_1^0| + |Q_2^0|/2}{|Q_1^0| + |Q_2^0| + |Q_1^2| + |M \cap E(b_1^2)|/2} \\
&\leq \frac{3|Q_1^1|/8 + |Q_1^2| + |Q_1^2|/2}{15|Q_1^1|/8 + |Q_2^0|} \\
&= \frac{4|Q_1^1| + 2|Q_2^0|}{15|Q_1^1| + 8|Q_2^0|} \\
&\leq \frac{4}{15}.
\end{align}
From inequality 2 to inequality 3, we use the fact that $|Q_1^0| \geq 3|Q_1^1|/8$.

**Case 2.** $|Q_1^0| < 3|Q_1^1|/8$. For this case, we put $v$ into the minimum vertex cover of $G$. We will delete all edges in $Q_0^1$ and some edges in $Q_2^0$ to make $G[V(b_1^2) \cup \{u\}]$ be a König subgraph. Since $|E(G[A_1])| + |E(G[A_2])| \leq |Q_0^1|/2$, one value of $|E(G[A_1])|$ and $|E(G[A_2])|$ is at most $|Q_0^1|/4$. Without loss of generality, assume that $|E(G[A_1])| \leq |Q_0^1|/4$. We put the vertices in $A_2$ into the minimum vertex cover of $G$, delete all the edges in $Q_0^1$, and delete the edges in $E(G[A_1])$ from subgraph $G[V(b_1^2) \cup \{u\}]$. Let $G'$ be the new subgraph obtained. It is easy to see that the size of minimum vertex cover in $G'$ is $|A_2| + 1 = (V(b_1^2) - 1)/2 + 1$. Since $uv \in M$ and $|M \cap E(G[V(b_1^2) \cup \{u\})| = (V(b_1^2) - 1)/2 + 1$, subgraph $G'$ is a König subgraph. We now analyze the proportion of the deleted edges in $Q_0^1$ and $G[V(b_1^2) \cup \{u\}]$.
\begin{align}
\frac{|Q_0^1| + |E(G[A_1])|}{|Q_0^1| + |Q_2^0| + |Q_1^2| + |M \cap E(b_1^2)| + 1} &\leq \frac{|Q_0^1| + |Q_2^0|/4}{|Q_0^1| + |Q_2^0| + |Q_1^2| + |M \cap E(b_1^2)|/4} \\
&\leq \frac{3|Q_1^1|/8 + |Q_2^0|/4}{3|Q_1^1|/2 + |Q_2^0|} \\
&= 1/4.
\end{align}
New Algorithms for Edge Induced König-Egerváry Subgraph

From inequality 4 to inequality 5, we use the fact that $|Q_1^h| < 3|Q_1^h|/8$. If $Y \setminus U$ is not an empty set, then for any vertex $u'$ in $Y \setminus U$, an isolated vertex $b_1$ in $B_1$ can be found such that the edge formed by $u'$ and $b_1$ is in maximum matching $M$. For this case, we put the vertices in $Y \setminus U$ into the minimum vertex cover. It is easy to get that the subgraph $G[V(B_1) \cup (Y \setminus U)]$ is a König subgraph.

For the case when $B_2$ is an empty set, it is obvious to get that $Y \setminus U$ is not empty, which can be handled as above.

Therefore, after dealing with all connected components of $B_2$ and $B_1$, a König subgraph $G'_2$ in $G_2$ can be found such that $|M'| = |M \cap E(G_2)| = |Y| + |M \cap E(B_2)|$, and $|E(G'_2)| \geq 11|E(G_2)|/15$.

We now deal with the connected components of $B_3$.

Lemma 7. Let $G_3$ be the graph constructed by the subgraph $G[V(B_3)]$ and edges in $SE$. Then, a König subgraph $G'_3$ can be obtained in $G_3$ such that $|M'| = |E(G_3) \cap M|$, and $|E(G'_3)| \geq 2|E(G_3)|/3$, where $M'$ is a maximum matching in $G'_3$. If graph $G$ contains no $C_3$ as connected component, then $|E(G'_3)| \geq 3|E(G_3)|/4$.

Proof. For the case when $B_3$ is an empty set, the correctness of the lemma is trivial. Assume that $B_3$ is not empty. Let $B_3 = \{b_1^3, \ldots, b_{h_3}^3\}$. For each connected component $b_i^3$ ($1 \leq i \leq h_3$) in $B_3$, if $b_i^3$ is a $C_3$ and a connected component in graph $G$, then no vertex in $b_i^3$ is connected to vertices in $V$, and there exists a König subgraph in $b_i^3$ with $2|E(b_i^3)|/3$ number of edges. On the other hand, if $b_i^3$ is a $C_3$ in $G[X]$ and not a connected component in graph $G$, then there exists a special edge $e$ in $SE$ with one endpoint in $b_i^3$, and a candidate deleted edge is contained in $b_i^3$. In the process of dealing with the connected components of $B_2$, all the edges in $E(Y, b_i^3)$ except special edge $e$ are handled, i.e., the edges in $E(Y, b_i^3) \setminus \{e\}$ are either covered by the vertices in $Y$, or not contained in the König subgraph. For this case, we delete such edges, and put the endpoint of special edge $e$ in $b_i^3$ into the minimum vertex cover. Let $G'_3$ be the graph constructed by the subgraph $G[V(b_i^3)]$ and special edge $e$. Thus, a König subgraph $G'$ of graph $G'_3$ can be obtained. The proportion of the deleted edges in $G'_3$ to get the König subgraph $G'$ is $1/4$. Thus, after dealing with all connected components of $B_3$, a König subgraph $G'_3$ can be found in $G_3$ such that $|M'| = |E(G_3) \cap M|$, and $|E(G'_3)| \geq 2|E(G_3)|/3$. If graph $G$ contains no $C_3$ as connected component, then $|E(G'_3)| \geq 3|E(G_3)|/4$.

For any connected component $C$ of $B_3$, assume that $C$ is also a connected component in $G$. Then, $C$ is a triangle in $G$. It is easy to see that two edges of a $C$ can be in edge induced König subgraph of $C$, and any edge of $C$ can be deleted to get the edge induced König subgraph. Therefore, for any given instance $(G = (V, E), k)$ of the EDGE INDUCED KÖNIG SUBGRAPH problem, we can firstly deal with the $C_3$ copies in graph $G$, without having any impact on the approximation ratio of the problem. We now give a refined analysis for the results in Lemma 7.

Lemma 8. Let $G_3$ be the graph constructed by the subgraph $G[V(B_3)]$ and edges in $SE$, where no connected component in $B_3$ is a connected component in $G$. Then, a König subgraph $G'_3$ can be obtained in $G_3$ such that $|M'| = |E(G_3) \cap M|$, and $|E(G'_3)| \geq 3|E(G_3)|/4$, where $M'$ is a maximum matching in $G'_3$.

We now study the properties of the connected components of $B_5$. Assume that $B_5$ is not empty, and let $B_5 = \{b_1^5, \ldots, b_{h_5}^5\}$.

Lemma 9. For any connected component $b_i^5$ ($1 \leq i \leq h_5$) of $B_5$, if $b_i^5$ is a hole, then a König subgraph with at least $4|E(b_i^5)|/5$ edges can be obtained.
Proof. Assume that \( b^5_i \) is a hole. Since hole \( b^5_i \) is factor-critical, it contains at least five edges. By deleting any edge in \( b^5_i \), a König subgraph of \( b^5_i \) can be obtained, and contains \(|E(b^5_i)|-1\) edges. Thus, if \( b^5_i \) is a hole, then a König subgraph with at least \( 4|E(b^5_i)|/5 \) edges can be obtained.

Lemma 10. For any subgraph \( C \) in \( G[X] \), if \( C \) is factor-critical, then each vertex in \( C \) has degree at least two in \( C \).

Proof. Assume that \( C \) is factor-critical. Then, for any vertex \( v \) in \( C \), \( C\setminus\{v\} \) has a perfect matching. It is easy to see that \(|E(C)| \leq |E(C')|/2 \). Let \( M = (M - M') \cup M'' \). Let \( W_3 = E(w, V(b^5_i)\setminus\{w\}) \), and \( W_4 = E(G[V(b^5_i)\setminus\{w\}]\setminus M) \).

Lemma 11. For each connected component \( b^5_i \) (1 ≤ \( i \) ≤ \( h_5 \)) of \( B_b \), if \( b^5_i \) is not a hole, then \(|W_2| \leq |W_4|^2\).

Lemma 12. Let \( G_4 \) be the subgraph induced by vertices in \( B_b \). Then, there exists a König subgraph \( G' \) such that \(|M'| = |M \cap E(G_4)|\), and \(|E(G_4)| \geq 7|E(G_4)|/10 \), where \( M' \) is a maximum matching in \( G_4' \).

Proof. For any connected component \( b^5_i \) (1 ≤ \( i \) ≤ \( h_5 \)) of \( B_b \), if \( b^5_i \) is a hole, then by Lemma 9, there exists a König subgraph \( G' \) in \( G[V(b^5_i)] \) with \(|E(G')| \geq 4|E(b^5_i)|/5 \). Now assume that \( b^5_i \) is not a hole. By Lemma 5, a bisection \((A_3, A_4)\) of size at least \(|m''/2 + |E(b^5_i) \cap M|/2| \) in subgraph \( G[V(b^5_i)\setminus\{w\}] \) can be found in \( O(m'' + |V(b^5_i)\setminus\{w\}|) \) time, where \( m'' \) is the number of edges in \( G[V(b^5_i)\setminus\{w\}] \). Since \( m'' = |E(b^5_i) \cap M| + |W_2| \), we get that the number of cut edges of bisection \((A_3, A_4)\) is at least \(|E(b^5_i) \cap M| + |W_2|/2 \). It is easy to get that \(|E(G[A_3])| + |E(G[A_4])| \leq |W_2|^2/2 \). We have \(|E(w, A_3)| + |E(G[A_3])| + |E(w, A_4)| + |E(G[A_4])| \leq |W_1| + |W_2|^2/2 \), and one value of \(|E(w, A_3)| + |E(G[A_3])|\) and \(|E(w, A_4)| + |E(G[A_4])|\) is at most \(|W_1|^2 + |W_2|^2 + |W_3|^2 + |W_4|^2/2 \). Without loss of generality, assume that \(|E(w, A_3)| + |E(G[A_3])| \leq (|W_1|^2 + |W_2|^2 + |W_3|^2 + |W_4|^2)/2 \). Delete the edges in \( E(w, A_3) \cup E(G[A_3]) \) from subgraph \( G[V(b^5_i) \cup \{w\}] \), and let \( G' \) be the resulted subgraph, which is a König subgraph by Lemma 2. We now analyze the proportion of the deleted edges in \( b^5_i \).

\[
\frac{|E(w, A_3)| + |E(G[A_3])|}{|E(b^5_i)|} \leq \frac{(|W_1|^2 + |W_2|^2 + |M \cap E(b^5_i)|)}{|W_1|^2 + |W_2|^2 + |M \cap E(b^5_i)|/2} \quad (6)
\]

\[
\frac{(|W_1|^2 + |W_2|^2 + |M \cap E(b^5_i)|)}{|W_1|^2 + |W_2|^2 + |M \cap E(b^5_i)|/2} \leq \frac{(|W_1|^2 + |W_2|^2 + |W_1|^2)/2}{|W_1|^2 + |W_2|^2 + |W_1|^2/2} \quad (7)
\]

\[
\frac{|W_1|^2/2 + |W_2|^2/4}{3|W_1|^2/2 + 3|W_2|^2/4 + |W_1|^2/6 + |W_2|^2/12} \quad (8)
\]

\[
\frac{|W_1|^2/2 + |W_2|^2/4}{3|W_1|^2/2 + 3|W_2|^2/4 + |W_1|^2/6 + |W_2|^2/12} \leq 3/10. \quad (9)
\]
From inequality 6 to inequality 7, we use the fact that $|M \cap E(b_i^1)| \geq |W_i^1|/2$. Inequality 9 is obtained from inequality 8 by Lemma 11.

By Lemma 4, Lemma 6, Lemma 7, and Lemma 12, we get the following result.

\textbf{Theorem 13.} For a given graph $G = (V, E)$, there exists an edge induced König subgraph $G'$ of $G$ such that $G'$ contains at least $2|E|/3$ edges.

By Lemma 4, Lemma 6, Lemma 12, and Lemma 8, we get the following result.

\textbf{Theorem 14.} For the Edge Induced König Subgraph problem, an approximation algorithm with ratio $10/7$ can be obtained in polynomial time.

4 Kernelization for Edge Induced König Subgraph above Lower Bound

For the Edge Induced König Subgraph problem, using the results in Theorem 13, it is easy to get a kernel with at most $3k/2$ edges for the problem. In other words, if $2m/3 > k$, then the given instance is a Yes-instance. Otherwise, we have $m \leq 3k/2$. Under this parameterization, $k$ is not a small value. In this paper, we study the following problem.

**Edge Induced König Subgraph above lower bound:**

Given a graph $G = (V, E)$ and non-negative integer $k$, find a set of at least $\lceil 2m/3 \rceil + k$ edges that induce a König subgraph, or report that no such set exists, where $m$ is the number of edges in $G$.

For a given instance $(G, k)$ of the Edge Induced König Subgraph above lower bound problem, we give the following two reduction rules.

Rule 1. For each connected component $C$ of $G$, if $C$ is a $C_3$, then remove $C$ from $G$.

Rule 2. For each connected component $C$ of $G$, if $C$ is a tree, then remove $C$, and $k = k - |E(C)|/3$.

\textbf{Lemma 15.} Rule 1 is correct and can be executed in polynomial time.

\textbf{Lemma 16.} Rule 2 is correct and can be executed in polynomial time.

\textbf{Theorem 17.} The Edge Induced König Subgraph above lower bound problem admits a kernel of $30k$ edges.

\textbf{References}


New Algorithms for Edge Induced König-Egerváry Subgraph


Computing Approximate Statistical Discrepancy

Michael Matheny
University of Utah, Salt Lake City, USA
mmath@cs.utah.edu

Jeff M. Phillips
University of Utah, Salt Lake City, USA
jeffp@cs.utah.edu

Abstract
Consider a geometric range space \((X,A)\) where \(X\) is comprised of the union of a red set \(R\) and blue set \(B\). Let \(\Phi(A)\) define the absolute difference between the fraction of red and fraction of blue points which fall in the range \(A\). The maximum discrepancy range \(A^* = \arg \max_{A \in (X,A)} \Phi(A)\). Our goal is to find some \(\hat{A} \in (X,A)\) such that \(\Phi(A^*) - \Phi(\hat{A}) \leq \varepsilon\). We develop general algorithms for this approximation problem for range spaces with bounded VC-dimension, as well as significant improvements for specific geometric range spaces defined by balls, halfspaces, and axis-aligned rectangles. This problem has direct applications in discrepancy evaluation and classification, and we also show an improved reduction to a class of problems in spatial scan statistics.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases Scan Statistics, Discrepancy, Rectangles

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.32

Related Version A full version of the paper is available at [17], https://arxiv.org/abs/1804.11287.

Acknowledgements Meg Rosales for her editing and Praful Agarwal for many discussions.

1 Introduction

Let \(X\) be a set of \(m\) points in \(\mathbb{R}^d\) for constant \(d\). Let \(X = R \cup B\) be the union (possibly not disjoint) of two sets \(R\), the red set, and \(B\), the blue set. Also consider an associated range space \((X,A)\); we are particularly interested in range spaces defined by geometric shapes such as rectangles in \(\mathbb{R}^d\) \((X,R_d)\), disks in \(\mathbb{R}^2\) \((X,D)\), and \(d\)-dimensional halfspaces \((X,H_d)\).

Let \(\mu_R(A) = |R \cap A|/|R|\) and \(\mu_B(A) = |B \cap A|/|B|\) be the fraction of red or blue points, respectively, in the range \(A\). We study the discrepancy function \(\Phi_X(A) = |\mu_R(A) - \mu_B(A)|\), when for brevity is typically write as just \(\Phi(A)\). A typical goal is to compute the range \(A^* = \arg \max_{A \in (X,A)} \Phi(A)\) and value \(\Phi^* = \Phi(A^*)\) that maximizes the given function \(\Phi\). Our goal is to find a range \(\hat{A}_\varepsilon\) that satisfies \(\Phi(\hat{A}_\varepsilon) \geq \Phi^* - \varepsilon\).

The exact version of this problem arises in many scenarios, formally as the classic discrepancy maximization problem [3, 7]. The rectangle version is a core subroutine in algorithms ranging from computer graphics [8] to association rules in data mining [9]. Also, for instance, in the world of discrepancy theory [20, 6], this is the task of evaluating how large the discrepancy for a given coloring is. For the halfspace setting, this maps to the minimum disagreement problem in machine learning (i.e., building a linear classifier) [16]. When \(\Phi\) is

---

1 Thanks to supported by NSF CCF-1350888, IIS-1251019, ACI-1443046, CNS-1514520, and CNS-1564287.
replaced with a statistically motivated form [12, 13], then this task (typically focusing on disks or rectangles) is the core subroutine in the GIScience goal of computing the spatial scan statistic [11, 22, 2, 1] to identify spatial anomalies. Indeed this statistical problem can be reduced the approximate variant with the simple discrepancy maximization form [2].

The approximate versions of these problems are often just as useful. Low-discrepancy colorings [20, 6] are often used to create the associated ε-approximations of range spaces, so an approximate evaluation is typically as good. It is common in machine learning to allow ε classification error. In spatial scan statistics, the approximate versions are as statistically powerful as the exact version and significantly more scalable [19].

While the exact versions take super-linear polynomial time in m, e.g., the rectangle version with linear functions takes Ω(m^2) time conditional on a result of Backurs et al. [3], we show approximation algorithms with O(m + poly(1/ε)) runtime. This improvement is imperative when considering massive spatial data, such as geotagged social media, road networks, wildlife sightings, or population/census data. In each case the size m can reach into the 100s of millions.

While most prior work has focused on improving the polynomials on the exact algorithms for various shapes [14, 25] or on using heuristics to ignore regions [28, 22], little work exists on approximate versions. These include [1] which introduced generic sampling bounds, [19] which showed that a two-stage random sampling can provide some error guarantees, and [27] which showed approximation guarantees under the Bernoulli model. In this paper, we apply a variety of techniques from combinatorial geometry to produce significantly faster algorithms; see Table 1.

**Our results.** Our work involves constructing a two-part coreset of the initial range space (X, A); it approximates the ground set X and the set of ranges A. This needs to be done in a way so that ranges can still be effectively enumerated and µR(A) and µB(A) values tabulated. We develop fast coreset constructions, and then extend and adapt exact scanning algorithms to the sparsified range space.

We develop notation and review known solutions in Section 2; also see Table 1. Then we describe a general sampling result in Section 3 for ranges with bounded VC-dimension. In particular, many of these results can be seen as formalizations and refinements (in theory and practice) of the two-stage random sampling ideas introduced in [19].

In Section 3.1 we describe improvements for halfspaces and disks. The details, defer to the full version [17], first improve upon the sampling analysis to approximate ranges H^2. By carefully annotating and traversing the dual arrangement from the approximate range space, we improve further upon the general construction.

Then in Section 4 we describe our improved results for rectangles. We significantly extend the exact algorithm of Barbay et al. [4] and obtain an algorithm that takes O(m + 1/ε log 1/ε). This is improved to O(m + 1/ε log log 1/ε) with some more careful analysis in the full version [17]. This nearly matches a new conditional lower bound of Ω(m + 1/ε), assuming current algorithms for APSP are optimal [3].

In Section 5 we show how to approximate a statistical discrepancy function (sdf, defined in Section 5) Φ, as well as any general function Φ. These require altered scanning approaches and the sdf-approximation requires a reduction to a number of calls to the generic ("linear") Φ. We reduce the number of needed calls to generic Φ functions from O(1/ε log 1/ε) [2] to O(1/√ε).

Finally, in Section 6 we show on rectangles strong empirical improvement over state of the art [19].
We highlight two general combinatorial properties of geometric range spaces. These are
to review, a range space \((X, A)\) is composed of a ground set \(X\) (for instance a set of points in \(\mathbb{R}^d\)) and a family of subsets \(A\) of that set. In this paper we are interested in geometrically defined range spaces \((X, A)\), where \(X \subset \mathbb{R}^d\). We formalize the requirements of this geometry via a conforming geometric mapping \(\psi\); see Figure 1. Specifically, it maps from a subset \(Y \subset X\) to subset of \(\mathbb{R}^d\). Typically, the result is a Lebesgue measurable subset of \(\mathbb{R}^d\), for instance \(\psi_D(Y)\), defined for disk range space \((X, D)\), could map to the smallest enclosing disk of \(Y\).

We say this mapping \(\psi_A\) is conforming to \(A\) if for any \(N \subset X\) it has the properties:

- for any subset \(A \in \mathcal{N}_A\) then \(\psi_A(A) \cap N = A\) \([\text{the mapping recovers the same subset}]\)
- for any subset \(Y \subset X\) then \(\psi_A(Y) \cap X \in (X, A)\) \([\text{the mapping is always in} (X, A)]\)

2 Background on Geometric Range Spaces

To review, a range space \((X, A)\) is composed of a ground set \(X\) (for instance a set of points in \(\mathbb{R}^d\)) and a family of subsets \(A\) of that set. In this paper we are interested in geometrically defined range spaces \((X, A)\), where \(X \subset \mathbb{R}^d\). We formalize the requirements of this geometry via a conforming geometric mapping \(\psi\); see Figure 1. Specifically, it maps from a subset \(Y \subset X\) to subset of \(\mathbb{R}^d\). Typically, the result is a Lebesgue measurable subset of \(\mathbb{R}^d\), for instance \(\psi_D(Y)\), defined for disk range space \((X, D)\), could map to the smallest enclosing disk of \(Y\).

We say this mapping \(\psi_A\) is conforming to \(A\) if for any \(N \subset X\) it has the properties:

- for any subset \(A \in \mathcal{N}_A\) then \(\psi_A(A) \cap N = A\) \([\text{the mapping recovers the same subset}]\)
- for any subset \(Y \subset X\) then \(\psi_A(Y) \cap X \in (X, A)\) \([\text{the mapping is always in} (X, A)]\)

2.1 Basic Combinatorial Properties of Geometric Range Spaces

We highlight two general combinatorial properties of geometric range spaces. These are critical in sparsification of the data and ranges, and enumeration of the ranges.

**Sparsification.** An \(\varepsilon\)-sample \(S \subset X\) of a range space \((X, A)\) preserves the density for all ranges as \(\max_{A \in \mathcal{A}} \frac{|X \cap A|}{|A|} = \frac{|(X \cap S) \cap A|}{|S \cap A|} \leq \varepsilon\). An \(\varepsilon\)-net \(N \subset X\) of a range space \((X, A)\) hits large ranges, specifically for all ranges \(A \in \mathcal{A}\) such that \(|X \cap A| \geq \varepsilon |X|\) we guarantee that \(N \cap A \neq \emptyset\). Consider range space \((X, A)\) with VC-dimension \(\nu\). Then a random sample \(S \subset X\) of size \(O\left(\frac{1}{\varepsilon^2}(\nu + \log \frac{1}{\varepsilon})\right)\) is an \(\varepsilon\)-sample with probability at least \(1 - \delta\) [26, 15]. Also a random sample \(N \subset X\) of size \(O\left(\frac{4}{\varepsilon^2} \log \frac{1}{\varepsilon^2}\right)\) is an \(\varepsilon\)-net with probability at least \(1 - \delta\). For our ranges of interest, the VC-dimensions of \((X, \mathcal{H}_d)\), \((X, D)\), and \((X, \mathcal{R}_d)\) are \(d\), \(3\), and \(2d\).

**Enumeration.** For the ranges spaces we will consider that each range can be defined by a basis \(B\); where \(B\) is a point set. Given a geometric conforming map \(\psi\) and subset \(Y\), a range space’s basis \(B \subset Y\) is such that \(\psi(B) = \psi(Y)\), but on a strict subset \(B' \subset B\), then \(\psi(B')\)
is different (and usually smaller under some measure) than \(\psi(B)\). We will use \(\beta\) to denote the maximum size of the basis for any subset \(Y \subset X\). For instance for \(\psi_D\) then \(\beta = 3\), for \(\psi_{R_d}\) then \(\beta = 2d\), and for \(\psi_{H_d}\) then \(\beta = d\). Recall, by Sauer’s Lemma [23], if a range space \((X, \mathcal{A})\) has VC-dimension \(\nu\), then \(\beta \leq \nu\).

This implies that for \(m = |X|\) points, there are at most \(\binom{m}{\beta} = O(m^\beta)\) different ranges to consider. We assume \(\beta\) is constant; then it is possible to construct \(\psi(Y)\) in \(O(|Y|)\) time, and to determine if \(\psi(Y)\) contains a point \(x \in X\) in \(O(1)\) time. This means we can enumerate all \(O(m^\beta)\) possible bases in \(O(m^\beta)\) time, construct their maps \(\psi(B)\) in as much time, and for all of them count which points are inside, and evaluate each \(\Phi(A)\) to find \(A^*\), in \(O(m^{\beta+1})\) time.

For the specific range spaces we study, the time to find \(A^* \in \mathcal{A}\) can be improved by faster enumeration techniques. For \(\mathcal{H}_d\), Dobkin and Eppstein [7] reduced the runtime to find \(A^*\) from \(O(m^{\beta+1})\) to \(O(m^\beta)\); this implies for \(\mathcal{D}\) the runtime is reduced from \(O(m^4)\) to \(O(m^3)\). For \(\mathcal{R}_d\), Barbuy et al. [4] show how to find \(A^*\) in \(O(m^3)\) time; this was recently shown tight [3] in \(\mathbb{R}^2\), assuming APSP takes cubic time.

### 2.2 Coverings

Our main approach towards efficient approximate range maximization, is to sparsify the range space \((X, \mathcal{A})\). This will have two parts. The first is simply replacing \(X\) with an \(\varepsilon\)-sample. The second is sparsifying the ranges \(\mathcal{A}\), using a concept we refer to as an \(\varepsilon\)-covering.

Recall that the symmetric difference of two sets \(A \Delta B = (A \cup B) \setminus (A \cap B)\). Define an \(\varepsilon\)-covering \((X, \mathcal{A}_\Delta)\) of a range space \((X, \mathcal{A})\) where \((X, \mathcal{A}_\Delta) \subset (X, \mathcal{A})\), so that for any \(A \in \mathcal{A}\) there exists a \(A' \in \mathcal{A}_\Delta\) such that \(|A \Delta A'| \leq \varepsilon|X|\). See Figure 2 for an illustration of this concept. If a range space satisfies the above condition for any one specific range \(A\), but not necessarily all ranges \(A \in \mathcal{A}\) simultaneously, then it is a weak \(\varepsilon\)-covering of \((X, \mathcal{A})\).

We will use subsets of the ground set to define subsets of the ranges. For a subset \(N \subset X\), let \(\mathcal{A}_{\mid N} = \{A \cap N \mid A \in \mathcal{A}\}\) be the restriction of \(\mathcal{A}\) to the points in \(N\). We will define \((X, \mathcal{A}_\Delta)\) using \(\mathcal{A}_{\mid N}\) or a subset thereof. However, as each \(A \in \mathcal{A}_{\mid N}\) is a subset of \(N\), which itself is a subset of \(X\), we need a conforming map \(\psi_A\) to take a region \(A \in \mathcal{A}_{\Delta}\) and map it back to some region in \(\mathcal{A}\), a subset of \(X\). Given \(\mathcal{A}'_{\mid N}\) (which is \(\mathcal{A}_{\mid N}\) or a subset) we define \((X, \mathcal{A}_\Delta)\) as

\[
(X, \mathcal{A}_\Delta) = \{X \cap \psi_A(A) \mid A \in \{N, \mathcal{A}'_{\mid N}\}\}.
\]

A small sized \(\varepsilon\)-covering is implied by a result of Haussler [10]. For every range space \((X, \mathcal{A})\) of VC-dimension \(\nu\), with \(m = |X|\), there always exist a maximal set of ranges \(\mathcal{A}_\Delta\) of size \(O((\frac{m}{1+\varepsilon})^\nu)\) where for every pair of ranges \(A, A' \in \mathcal{A}_\Delta\) the symmetric difference \(|A \Delta A'| \geq k\). Setting \(k = m\varepsilon\) then \((\frac{m}{1+\varepsilon})^\nu = O(\frac{1}{\varepsilon^\nu})\), so \(\mathcal{A}_\Delta\) is an \(\varepsilon\)-covering.
Symmetric difference nets. We can construct an $\varepsilon$-net over the symmetric difference range space of $A$ and then use these points to define $A_\Delta$.

For a family of ranges $A$, let $S_A$ be the family of ranges made up of the symmetric difference of ranges of $A$. Specifically $S_A = \{A_1 \Delta A_2 \mid A_1, A_2 \in A\}$. If range space $(X, A)$ has VC-dimension $\nu$, then $(X, S_A)$ has VC-dimension at most $O(\nu \log \nu)$ [21]. Thus for constant $\nu$ we can use asymptotically the same size random sample as before. Matheny et al. [19] pointed out two important properties connecting nets over symmetric difference range spaces and $\varepsilon$-coverings and then finding $\hat{A}_\varepsilon$.

(P1) An $\varepsilon$-net $N$ for $(X, S_A)$ induces $(N, A_{|N})$ which is an $\varepsilon$-covering of $(X, A)$ [19].

(P2) Given an $\varepsilon$-covering $(N, A_{|N})$ and an $\varepsilon$-sample $S$ over $(X, A)$ then for any range $A \in (X, A)$, there exists a range $\psi_A(A') \cap X$ for $A' \in A_{|N}$ so that $\left| \frac{|A' \cap X|}{|X|} - \frac{|\psi_A(A') \cap S|}{|S|}\right| \leq \varepsilon$ [19].

For an appropriate constant $C$, by constructing $(\varepsilon/C)$-nets $N_R$ and $N_B$, of size $n$, on the red $(R, S_A)$ and blue $(B, S_A)$ points, also constructing $(\varepsilon/C)$-samples of size $s$ on $(R, A)$ and $(B, A)$, and invoking (P2) on the results, Matheny et al. [19] observed we can maximize $\Phi(\psi_A(A') \cap S)$ over $A' \in A_{|N_R} \cup A_{|N_B}$ to find an $\varepsilon$-approximate $\hat{A}_\varepsilon$. They construct the $\varepsilon$-nets and $\varepsilon$-samples using random sampling, and apply the results to scan disk $D$ and rectangle $R_2$ range spaces towards finding $\hat{A}_\varepsilon$. Enumerating all ranges in $A' \in A_{|N_R} \cup A_{|N_B}$ and counting the intersections with the $(\varepsilon/C)$-samples, when $C$ is a constant, is sufficient to find an $\hat{A}_\varepsilon$ in time $O(m + |N|^2 |S| \log n) = O(m + \frac{1}{\varepsilon} \log^3 \frac{1}{\varepsilon})$ for disks $(X, D)$ and time $O(m + |N|^4 + |S| \log n) = O(m + \frac{1}{\varepsilon} \log^3 \frac{1}{\varepsilon})$ for rectangles $(X, R_2)$.

We can ignore the distinct red and blue points, and focus on three aspects of this problem which can be further optimized: (1) More efficiently constructing a sparse set of $\varepsilon$-covering ranges $(X, A_\Delta)$. (2) More efficiently constructing a smaller $\varepsilon$-sample $S$ of $(X, A)$. (3) More efficiently scanning the resulting $(S, A_\Delta)$.

3 General Results via $\varepsilon$-Coverings

For general range spaces of constant VC-dimension $\nu$ we can directly apply the work of Matheny et al. [19] to get a bound. A random sample $N$ of size $O(\frac{\nu \log \nu}{\varepsilon^2} \log \frac{1}{\varepsilon})$ induces an $\varepsilon$-covering $(X, A_{|N})$ with constant probability by (P1). A random sample $S$ of size $O(\frac{\nu}{\varepsilon^2})$ induces an $\varepsilon$-sample with constant probability. By (P2), scanning the ranges in $(X, A_{|N})$, evaluating $\Phi(A)$ on each ranges $A$ using $S$, and returning the maximum $\hat{A}_\varepsilon$ induces the $\varepsilon$-approximation of $\Phi(A^*)$ as we desire. Including the time to calculate $N$ and $S$ we obtain the following result.

Theorem 1. Consider a range space $(X, A)$ with constant VC-dimension $\nu$, with $|X| = m$, and conforming map $\psi_A$. For $A^* = \arg \max_{A \in A} \Phi(A)$, with probability at least $1 - \delta$, in time $O\left( m + \frac{1}{\varepsilon} \log ^3 \frac{1}{\varepsilon} \log \frac{1}{\delta}\right)$, we can find a range $\hat{A}_\varepsilon$ so that $|\Phi(A^*) - \Phi(\hat{A}_\varepsilon)| \leq \varepsilon$.

Proof. First compute random samples $N$ and $S$ of size $O(\frac{1}{\varepsilon} \log \frac{1}{\delta})$ and $O(\frac{\nu}{\varepsilon^2})$ respectively. The algorithm naively considers all $O((\frac{1}{\varepsilon} \log \frac{1}{\delta})^\nu)$ subsets $B \subset N$ of size $\nu$, and calculates the quantity $\Phi(S \cap \psi_A(B))$. By (P2), this can be used to $\varepsilon$-approximate $\Phi(A)$ for any range $A \in A$ which has less than $\varepsilon$-symmetric difference with $\psi_A(B)$. Moreover, since $(X, A_{|N})$ is an $\varepsilon$-cover, with constant probability any range $A$ is within symmetric difference of at most $\varepsilon m n$ of one induced by some subset $B$. Thus, with constant probability we observe some range $\hat{A}_\varepsilon = X \cap \psi_A(B)$ for which $|\Phi(A^*) - \Phi(\hat{A}_\varepsilon)| \leq \varepsilon$ (after adjusting constants in the size of $N$ and $S$). To amplify the probability of success to $1 - \delta$, we repeat this process $O(\log \frac{1}{\delta})$ times, and return the $\hat{A}_\varepsilon$ with median score. ▶
3.1 Halfspaces

The above general result applied to halfspaces \((X, \mathcal{H}_d)\), would require \(O(m + \frac{1}{\varepsilon^2} \log d \frac{1}{\varepsilon} \log \frac{1}{\varepsilon})\) time. We improve this runtime to \(O(m + \frac{1}{\varepsilon^2} \log \frac{1}{\varepsilon})\). First, a recent paper [18] shows that with constant probability an \(\varepsilon\)-sample \(S\) for \((X, \mathcal{H}_2)\) of size \(s = O\left(\frac{1}{\varepsilon^2} \log^{2/3} \frac{1}{\varepsilon}\right)\) can be constructed in \(O(m + \frac{1}{\varepsilon^2} \log \frac{1}{\varepsilon})\) time. Second we create a weak \(\varepsilon\)-covering of \((X, \mathcal{H}_d)\) using \((X, \mathcal{H}_d[N])\) for a random sample \(N\). We show this only requires a random sample of size \(O\left(\frac{d}{\varepsilon^2} \log d\right)\) = \(O(1/\varepsilon)\). Then, we show how to enumerate these ranges \((X, \mathcal{H}_d[N])\) while maintaining the counts from \(S\) (an \(\varepsilon\)-sample of only \((X, \mathcal{H}_2)\)) with less overhead than the previous brute force approaches. Ultimately this requires time \(O(m + \frac{1}{\varepsilon^{1.5}} \log^{2/3} \frac{1}{\varepsilon})\), with constant probability. For space, the details are in the full version [17].

Moreover, this can be applied to disks \((X, \mathcal{D})\) in \(O(m + \frac{1}{\varepsilon^{1.5}} \log^{2/3} \frac{1}{\varepsilon})\) time.

4 Rectangles

For the case of rectangles \((X, \mathcal{R}_d)\), we will describe two classes of algorithms. One simply creates an \(\varepsilon\)-cover \((X, \mathcal{R}_d[N])\) and evaluates each rectangle \(A\) in this cover on an \(\varepsilon\)-sample \(S\) as before. The other takes specific advantage of the orthogonal structure of the rectangles and of “linearity” of \(\Phi\); this algorithm can find the maximum in \(\Phi\) among ranges in \((X, \mathcal{R}_d[N])\) without considering every possible range. Our techniques are inspired by several algorithms [4, 24, 8] for the exact maximization problem, but requires new ideas to efficiently take advantage of using both \(N\) and \(S\). Common to all techniques will be an efficient way to compute an \(\varepsilon\)-cover based on a grid.

Grid \(\varepsilon\)-covers for rectangles. We create a grid \(G\) defined as the cross-product of \(r = O(1/\varepsilon)\) cells along each axis. Straightforward details of its construction and use are in the full version [17]. We label the rectangular ranges of \(X\) restricted to this grid boundary as \((X, \mathcal{R}_d[G])\); it is an \(\varepsilon\)-cover of \((X, \mathcal{R}_d)\). The main results of this \(\varepsilon\)-cover are in the next lemma and theorem.

- **Lemma 2.** For range space \((X, \mathcal{R}_d)\) where \(|X| = m\), the construction of grid \(G\) takes \(O(m \log m + \frac{1}{\varepsilon^2})\) time, has \(O(1/\varepsilon)\) cells on each side, and induces an \(\varepsilon\)-cover \((X, \mathcal{R}_d[G])\) of \((X, \mathcal{R}_d)\) for constant \(d > 1\).

- **Theorem 3.** Consider a range space \((X, \mathcal{R}_d)\) with \(|X| = m\) and an Lipschitz-continuous function \(\Phi\) with maximum range \(A^* = \arg \max_{A \in \mathcal{R}_d} \Phi(A)\). With probability at least \(1-1/\varepsilon^{1/\varepsilon}\), in time \(O(m + \frac{1}{\varepsilon^2})\) we can find a range \(A_\varepsilon\) so that \(|\Phi(A^*) - \Phi(A_\varepsilon)| \leq \varepsilon\).

4.1 Algorithms for Decomposable Functions

Here we exploit a critical “linear” property of \(\Phi\) that a rectangle \(A\) can be decomposed into any two parts \(A_1\) and \(A_2\) and \(\Phi(A) = \Phi(A_1) + \Phi(A_2)\). Technically, we solve both \(\Phi^+(A) = \mu_R(A) - \mu_B(A)\) and \(\Phi^-(A) = \mu_B(A) - \mu_R(A)\) separately, and take their max. In particular, this allows us (following exact algorithms [4]) to decompose the problem along a separating line. The solution then either lies completely on one half, or spans the line. In the exact case on \(s\) points, this ultimately leads to a run time recurrence of \(T_1(s) = 2T_1(s/2) + T_2(s)\) where \(T_2(s)\) is the time to compute the problem spanning the line. The line spanning problem can then be handled using a different recurrence that leads to \(T_2(s) = O(s^2)\) and a total runtime for the problem of \(T(s) = 2T_1(s/2) + O(s^2) = O(s^2)\) [4].

First we show we can efficiently construct a special sample \(S\) of size \(s = O(1/(\varepsilon^2 \log \varepsilon))\), but this still would requires runtime of roughly \(1/\varepsilon^4\).
Our approximate algorithm will significantly improve upon this by compressing the representation at various points, but requiring some extra bookkeeping and a bit more complicated recurrence to analyze. In short, we can map \( S \) to an \( r \times r \) grid (using Lemma 2), and then the recurrence only depends on the dyadic \( y \)-intervals of the grid. We can compress each such interval to have only \( \varepsilon s / \log r \) error, since each query only touches about \( \log r \) of these intervals. The challenge then falls to maintaining this compressed structure more efficiently during the recurrence.

The dense exact case on an \( r \times r \) grid is also well studied. There exists a practically efficient \( O(r^3) \) time method [5] based on Kadane’s algorithm (which performs best as \( \text{gridScan}_\text{linear} \); see Section 6), and a more complicated method taking \( O(r^3 (\log \log r \log r)^{1/2}) \) time [24]. By allowing an approximation, we ultimately reduce this runtime to \( O(r^2 \log r) = O(1/\varepsilon^2) \).

We will focus on the 2d case. This is where the advantage over the Theorem 3 bound of \( O(m + 1/\varepsilon^4) \) is most notable. Generalization to high dimensions is straightforward: enumerate over pairs of grid cells to define the first \( d - 2 \) dimensions, then apply the 2-dimensional result on the remaining dimensions.

**Tree and slab approximation.** The algorithm builds a binary tree over the rows (the \( y \) values) of \( G \). We will assume that the number of cells in each axis \( r = O(1/\varepsilon) \) is a power of 2 (otherwise we can round up), so it is a perfectly balanced binary tree.

At the \( i \)th level of the tree, each node contains \( r/2^i \) rows and there are \( 2^i \) nodes. We refer to the family of rows represented by a subtree as a *slab*. Any grid-aligned rectangle \( A = [x_1, x_2] \times [y_1, y_2] \) can be defined as the intersection of \( [x_1, x_2] \) with at most \( 2 \log_2 r \) slabs in the \( y \)-coordinate – the classic dyadic decomposition. This implies we can tolerate \( \eta s = O(\varepsilon s / \log r) \) additive error in each slab to have at most \( O(\varepsilon s) \) additive error overall (which implies the percentage of red and of blue points in each range has additive \( O(\varepsilon) \) error).

Since the rectangle will span the entire vertical extent (\( y \) direction) of each slab in this decomposition, the additive error of a slab can be obtained along just the horizontal (\( x \)) direction. Thus, we can scan cells from left to right within a slab, and only retain the cumulative weight in a cell when it exceeds \( \eta s \). We refer to this operation as \( \eta \)-compression. We denote each column (and \( x \) value) within a slab where it has retained a non-zero value as active, all other columns are inactive. We store the active cells in a linked list.

Since there are \( \Theta(s/r) \) points per row, it implies we can approximate each slab consisting of 1 row (a leaf of the tree, level \( \log_2 r \)) with weights in only \( O(1/(r \eta)) = O(\log r) \) cells (since \( r = O(1/\varepsilon) \)). And a slab at level \( i \) (originally with \( \Theta(s/2^i) \) points) can be approximated by accumulating weight in \( O(\min\{r, 1/(\eta 2^i)\}) \) cells. For level \( i > \log 1/\eta r \), this compresses the points in that slab.

> **Lemma 4.** In \( O(r^2) \) time, we can compress all slabs in the tree, so a slab at level \( i \) contains \( \ell_i = O(\min\{r, 1/(\eta 2^i)\}) \) active columns where \( \eta = O(\varepsilon /\log r) \).
Computing Approximate Statistical Discrepancy

Interval Preprocessing and Merging. Now consider a subproblem, where we seek to find a rectangle $A = [x_1, x_2] \times [y_1, y_2]$ to maximize the total weight, restricted to a given horizontal extent $[y_1, y_2]$ (e.g., within a slab). We reduce this to a 1d problem by summing the weights for each $x$-coordinate to $w_x = \sum_{y \in [y_1, y_2]} w_{x,y}$. Then there is an often-used [4, 7, 2] way to preprocess intervals $[x_1', x_2']$ so they can be merged and updated. It maintains 3 maximal weight subintervals: (1) the maximal weight subinterval in $[x_1', x_2']$, (2) the maximal weight interval including the left boundary $x_1'$, and (3) the maximal weight interval including the right boundary $x_2'$. Given two preprocessed adjacent intervals $[x_1', x_2']$ and $[x_2' + 1, x_3']$, we can update these subintervals to $[x_1', x_3']$ in $O(1)$ time. Thus given a horizontal extent with $a$ active intervals, we can find the maximum weight subinterval in $O(a)$ time.

Recursive construction. Now we can describe our recursive algorithm for finding the maximum is restricted to use their active columns. We sum weights on active columns needs to be maintained within $[x_1', x_2']$. A column of $M$ is active if it is active in $T$ or $B$. Counts in active columns of $M$ are maintained, and intervals of $M$ described by consecutive inactive columns have been merged. The goal is to find the maximum weight rectangle with vertical span $[y_1, y_2]$ where $y_2$ is in $T$ and $y_1$ is in $B$ (it must cross $M$).

We specifically want to solve this problem when $M$ is empty, $T$ is the top child and $B$ the bottom child of the root, and all columns are initially active. We call this the case of size $r$ since there are still $r$ rows.

Lemma 5. The Strip-constrained grid search problem of size $r$ over an $\eta$-compressed binary tree takes $O(r/\eta)$ time.

Proof. Following Barbay et al. [4] we split the problem into 4 subcases, following the subtrees of the slabs. Slab $T$ has a top $T_t$ and bottom $T_b$ sub-slab, and similarly $B_t$ and $B_b$ for $B$. Then we consider 4 recursive cases with new strip $M'$: (1) slabs $T_t$ and $B_b$ with $M' = T_b \cup M \cup B_t$, (2) slabs $T_b$ and $B_b$ with $M' = M \cup B_t$, (3) slabs $T_t$ and $B_t$ with $M' = T_b \cup M$, and (4) slabs $T_t$ and $B_t$ with $M' = M$. The cost in a recursive step is the preprocessing of the new slab $M'$. We will describe the largest case (1); the others are similar.

Strip $M$ already maintains preprocessed intervals of inactive columns. When $T_b$ or $B_t$ has an active column which is inactive in $T_t$ and $B_b$, we treat this as a new inactive interval that needs to be maintained within $M'$. The weights from $T_b$ and $B_t$ are added to that in the column for $M$. If inactive intervals of $M'$ are then adjacent to each other, they are merged, in $O(1)$ time each. This completes the recursive step for case (1).

In the base case when slabs $T$ and $B$ are single rows (at depth $O(\log r)$), the range maximum is restricted to use their active columns. We sum weights on active columns

\[
\begin{array}{c}
T \\
\hline
T_t & T_b \\
\hline
M \\
\hline
B \\
\hline
B_t & B_b
\end{array}
\]
in $T$, $B$, and $M$. Then also considering the inactive intervals on $M$, invoke the interval merging procedure [4] to find the maximal range, in time proportional to the number of active intervals, in $O(1/(2\log r)) = O(1/(\eta^2))$ time.

The cost of recursing in any case is also proportional to the number of active columns since this bounds the number of potential merges, and the time it takes to scan the linked lists of active columns to detect where the merging is needed. At level $i$ this is bounded by $t_i = \min\{r, 1/(\eta^2)\} \leq O(1/(\eta^2))$.

At each level $i$ there are $4^i$ recursive sub instances and at most $O(1/(2\eta))$ active columns, and therefore merging takes $Z_i = 4^iO(1/(2\eta)) = 2^iO(1/\eta)$ time. The cost is asymptotically dominated by the last level, which takes time $2^{\log_2 r}O(1/\eta) = O(r/\eta)$.

Letting $\eta = \varepsilon/(\log r) = O(1/(r\log r))$ (since $r = O(1/\varepsilon)$) as it is in Lemma 4 we have a bound of $T_2(r) = O(r^2 \log r)$. We can solve the first recurrence of $T_1(r) = 2T_1(r/2) + T_2(r) = 2T_1(r/2) + O(r^2 \log r) = O(r^2 \log r)$. Using $r = O(1/\varepsilon)$ this bounds the overall runtime of finding $\max_{R \in (X \cup (\mu \setminus R))} \Phi(R)$ as $O(1/\varepsilon \log 1/\varepsilon)$.

**Theorem 6.** Consider $(X, R_2)$ with $|X| = m$ and $A^* = \arg \max_{A \in R_2} \Phi(A)$. With probability at least $1 - \delta$, in time $O(m + \log \frac{1}{\varepsilon} \log \frac{1}{\delta})$, we can find a range $\hat{A}_\varepsilon$ so $|\Phi(A)^* - \Phi(\hat{A}_\varepsilon)| \leq \varepsilon$.

In the full version [17], we reduce this time to $O(m + \frac{1}{\varepsilon} \log \frac{1}{\varepsilon} \log \frac{1}{\delta})$.

For $(X, \mathcal{R}_d)$ and $d$ constant, the runtime increases to $O(m + \frac{1}{\varepsilon} \log \frac{1}{\varepsilon} \log \frac{1}{\delta})$.

**Conditional lower bound.** Backurs et al. [3] recently showed $\Omega(m^2)$ time is required to solve for $A^* = \arg \max_{A \in (X \cup \mathcal{R}_d)} \Phi(A)$, assuming that all pairs shortest path (APSP) requires cubic time. We can show this implies our algorithm is nearly tight. If we set $\varepsilon = 1/4m$ then if any algorithm could find an $\hat{A}_\varepsilon$ such that $\Phi(\hat{A}_\varepsilon) \geq \Phi(A^*) - \varepsilon$, then it would imply that $|\mu_R(A^*) - \mu_B(A^*)| - |\mu_R(\hat{A}) - \mu_B(\hat{A})| \leq \varepsilon$. And hence the difference in counts of points in each pair $\mu_R$ and $\mu_B$ is off by at most $2\varepsilon m = 2(1/4m)m = 1/2$. Thus it must be the optimal solution. If this can run in $o(m + 1/\varepsilon^2)$ time, it implies an $o(m^2)$ algorithm, which implies a subcubic algorithm for APSP, which is believed impossible.

**Theorem 7.** For $(X, \mathcal{R}_2)$ with $|X| = m$, and $A^* = \arg \max_{A \in \mathcal{R}_2} \Phi(A)$. It takes $\Omega(m + \frac{1}{\varepsilon^2})$ time to find a range $\hat{A}_\varepsilon$ so that $|\Phi(A)^* - \Phi(\hat{A}_\varepsilon)| \leq \varepsilon$, assuming APSP takes $\Omega(n^3)$ time.

## 5 Statistical Discrepancy Function Approximation

In this section we address approximating $\max_{A \in (X \cup \mathcal{A})} \Phi(A)$ when it is a more general function of $\mu_R(A)$, and $\mu_B(A)$. Rewrite $\Phi(A) = \phi(\mu_R(A), \mu_B(A))$, and in this section it will be more convenient to discuss $\phi(r, b)$ where $r = \mu_R(A)$ and $b = \mu_B(A)$.

We say $\phi$ is $(\tau, \gamma)$-linear if it can be represented with up to $\varepsilon$-error as the upper envelope of $\gamma$ functions of slope at most $\tau$. We can then simply maximize each function individually, and return the maximum overall score. When $\gamma$ and $\tau$ are constant (as with $\phi(r, b) = |r - b|$), we simply say the function is linear.

First observe that Theorem 1, algorithms in Section 3.1 (see full version [17]), and Theorem 3 simply evaluate $\Phi(A)$, so if this can be done in constant time, and the slope $\tau$ is constant, then these results automatically hold. However, Theorem 6 requires the linearity property.

For the spatial scan statistic application, the most common function [12] is defined $\phi_K(r, b) = r \ln \frac{r}{b} + (1 - r) \ln \frac{1 - r}{b}$, and is non-linear. We define a more general class of statistical discrepancy functions (SDF), which includes $\phi_K$. Such $\phi$ have domain $r, b \in [0, 1]$,
\( \phi(r, b) = 0 \) when \( r = b \) and this is its minimum, and \( \phi(r, b) \) is convex on \((0, 1)^2\). Moreover, for these functions, it suffices too consider a range \([\xi, 1 - \xi]^2\) for small constant \( \xi \) (c.f. [2, 1, 19]), and that in this range \( \phi \) is \( \tau \)-Lipschitz where \( \tau \) is a constant depending only \( \xi \).

Agarwal et al. [2] approximated such functions by considering \( O(\frac{1}{\tau} \log \frac{1}{\tau}) \) linear functions, each tangent to \( \phi \), so their upper envelope \( \hat{\phi} \) satisfied \( \max_{(r, b) \in [\xi, 1-\xi]^2} |\phi(r, b) - \hat{\phi}(r, b)| \leq \varepsilon \).

We will construct an approximation of \( \phi \) with linear functions with a very different approach. Unlike the previous approach which only considers the function \( \phi \), our approach adapts the set of linear functions to the function \( \phi \) and data \((X, A)\). It uses \( O(1/\sqrt{\varepsilon}) \) linear functions.

**Function approximation.** Consider the distinct ranges in \((X, A)\); each range \( A \) corresponds to a point \( p_A = (\mu_R(A), \mu_B(A)) \). Let \( P = \{ p_A \mid A \in (X, A) \} \) be this set of points. Then \( p_{A^*} \), must lie on \( CH(P) \), the convex hull of \( P \), where \( A^* = \arg \max_{A \in (X, A)} \Phi(A) \).

Moreover, each point \( p \) on \( CH(P) \) maximizes some linear function, \( f(r, b) = \alpha r + \beta b \). If \( p = \arg \max_{p' \in P} f(r_p, b_p) \), then it also maximizes \( f_o(r, b) = (\alpha/c)r + (\beta/c)b \) for any \( c > 0 \). We can therefore restrict our attention (by implicit choice of \( c \)) to only functions with \( \alpha^2 + \beta^2 = 1 \). These functions correspond to a dot product \( \langle(\alpha, \beta), (r, b)\rangle \) and are maximized by points on \( CH(P) \) where \((\alpha, \beta) \) is between two adjacent normals on the boundary of \( CH(P) \).

To further simplify, we now parameterize these functions by an angle \( \theta = \arccos(-\alpha) \) (where still \( \alpha^2 + \beta^2 = 1 \)). We focus on \( \theta \in [0, \pi/2] \) as we can always repeat the procedure on the other 3 quadrants.

Now let \( f_{\theta}^o \) be any linear function such that \( p_{A^*} = \arg \max_{p \in P} f_{\theta}^o(p) \) is maximized by the point \( p_{A^*} \) corresponding to the optimal range \( A^* \).

▶ **Lemma 8.** Consider \( p_1 = \arg \max_{p \in P} f_{\theta_1}(p) \) and \( p_2 = \arg \max_{p \in P} f_{\theta_2}(p) \) so that \( p_{A^*} = \arg \max_{p \in P} f_{\theta}^o(p) \) and \( \theta_1 \leq \theta \leq \theta_2 \). Then \( \phi(p_{A^*}) \leq \max\{\phi(p_1), \phi(p_2)\} + \tau \cdot \frac{||p_1-p_2||}{2} \tan \frac{\theta_2 - \theta_1}{2} \).

**Proof.** Define a triangle through points \( p_1, p_2, \) and a point \( p_3 \). The point \( p_3 \) is defined at the intersections of the normals to \( f_{\theta_1} \) at \( p_1 \) and to \( f_{\theta_2} \) at \( p_2 \). We refer to “above” in the normal direction of the edge between \( p_1 \) and \( p_2 \), and in the direction of \( p_3 \).

First we show that \( p_{A^*} \) must be inside the triangle. If it is above the edge connecting \( p_1 \) and \( p_3 \), then it would be \( \arg \max_{p \in P} f_{\theta_1}(p) \). Similarly it cannot be above the edge connecting \( p_2 \) and \( p_3 \). Also, it must be above the edge connecting \( p_1 \) and \( p_2 \), since otherwise by convexity \( \max(\phi(p_1), \phi(p_2)) > \phi(p_{A^*}) \) and one of \( p_1 \) or \( p_2 \) would maximize \( f_{\theta}^o \).

We say the height of the triangle \( h \) is defined as the distance from \( p_3 \) to \( q_3 \), where \( q_3 \) is the closest point on the edge through \( p_1 \) and \( p_2 \).

Let \( \angle_1 \) be the internal triangle angle at \( p_1 \), and \( \angle_2 \) at \( p_2 \). Then \( (\theta_2 - \theta_1) = \angle_1 + \angle_2 \).

Now \( h = ||p_1 - q_3|| \tan (\angle_1) = ||p_2 - q_3|| \tan (\angle_2) \) which, fixing \( ||p_1 - p_2|| \), is maximized when \( \angle_1 = \angle_2 = \frac{\theta_2 - \theta_1}{2} \). Summing \( h \leq ||p_1 - q_3|| \tan ((\theta_2 - \theta_1)/2) \) and \( h \leq ||p_2 - q_3|| \tan ((\theta_2 - \theta_1)/2) \) it can be seen that \( h \leq \frac{1}{2} ||p_1 - q_3|| + ||p_2 - q_3|| \tan ((\theta_2 - \theta_1)/2) = \frac{1}{2} ||p_1 - p_2|| \tan ((\theta_2 - \theta_1)/2) \).

Finally, we argue that \( \min\{\phi(p_{A^*}) - \phi(p_1), \phi(p_{A^*}) - \phi(p_2)\} \leq \tau \cdot h \). Let \( \gamma \) be the iso-curve
of $\phi$ at value $\phi(p_{A^*})$. It must pass above $p_1$ and $p_2$, otherwise they would be the maximum. It also must pass within a distance of $h$ from either $p_1$ or $p_2$ since $\gamma$ is convex, it contains $p_{A^*}$, and $p_{A^*}$ is within $b$ of the edge between $p_1$ and $p_2$. Then the lemma follows since $\phi$ is $\tau$-Lipschitz.

To choose a set of linear functions we start with two linear functions $f_0$ and $f_{\pi/2}$, whose maximum in $P$ are points $p_1$ and $p'_1$. These induce a triangle as in the proof of Lemma 8, and $p_{A^*}$ must be in this triangle. If its height $h = \frac{\|p_{A^*} - p_1\|}{2} \tan(\frac{\pi}{4}) > \epsilon/\tau$, then we choose a new function $f_{\pi/4}$ (at the midpoint of the two angles) whose maximum is point $p_2$. Now recurse on triangles defined by $p_1$ and $p_2$, and by $p_2$ and $p'_1$.

\textbf{Lemma 9.} The recursive algorithm considers at most $\sqrt{\tau/\epsilon}$ functions to maximize.

\textbf{Proof.} Index the points found by the algorithm $\{p_1, p_2, \ldots, p_{k+1}\}$ in the order they appear on the convex hull. Each consecutive pair $p_i$ and $p_{i+1}$ defines a triangle with height at most $\epsilon/\tau$. Let $\ell_i = \|p_i - p_{i+1}\|$ and $\gamma_i = \theta_{i+1} - \theta_i$ where the $p_i$ and $p_{i+1}$ where chosen by maximizing functions $f_{\theta_i}$ and $f_{\theta_{i+1}}$, respectively. It follows that $\sum_{i=1}^{k} \ell_i \leq 2$ and $\sum_{i=1}^{k} \gamma_i = \pi/2$. We also have for each triangle that $\frac{\pi}{\tau} \leq \frac{\ell_i}{\gamma_i} \tan(\frac{\pi}{4}) \leq \frac{\ell_i}{\gamma_i} \cdot \frac{2\pi}{\pi}$. Thus for each term we have $\ell_i \geq \frac{\tau}{2 \gamma_i}$, and summing over $k$ terms $\sum_{i=1}^{k} \frac{\pi}{\tau} \leq \sum_{i=1}^{k} \ell_i \leq 2$. Now in the inequality $\frac{\pi}{\tau} \geq \sum_{i=1}^{k} \frac{\ell_i}{\gamma_i}$ such that $\sum_{i=1}^{k} \gamma_i = \pi/2$, then $k$ is the largest when all of the $\gamma_i$ have the same value $\gamma_i = \frac{\pi}{2k}$. In this case, then $\frac{\pi}{\tau} \geq \sum_{i=1}^{k} \frac{\ell_i}{\gamma_i} = \sum_{i=1}^{k} \frac{\pi}{\tau} = k^2 \frac{\pi}{2 \tau}$. Solving for $k$ reveals $k \leq \sqrt{\epsilon/\tau}$.

Now we analyze the full algorithm for maximizing a statistical discrepancy function over $(X, R_d)$ with $\tau$ and $d$ as constants. We first invoke Lemma 2 to construct the grid in $O(m + \frac{1}{\tau} \log \frac{1}{\delta} \log \frac{1}{\delta} + \frac{1}{\tau})$ time. We then use Theorem 6 in $F = O(\frac{1}{\tau^2 \epsilon} \log \frac{1}{\delta})$ time to find the approximate maximum range for any linear function $\Phi'$.

Then we run the above recursive triangle algorithm repeatedly on the constructed grid, and each function maximization takes $F$ time. By Lemma 9 we need to make $O(\sqrt{1/\epsilon})$ calls. And by Lemma 8 one of the function calls must find an approximately correct answer.

\textbf{Theorem 10.} Consider a range space $(X, R_d)$ with $|X| = m$ and $d$ constant. For a statistical discrepancy function $\Phi$ with $\tau$ constant and with maximum range $A^* = \arg \max_{A \in R_d} \Phi(A)$, then with probability at least $1 - \delta$, in time $O(m + \frac{1}{\tau^2 \epsilon \tau^{1/2}} \log \frac{1}{\delta} + \frac{1}{\tau^2 \epsilon} \log \frac{1}{\delta} + \frac{1}{\tau^2})$, we can find a range $A_\epsilon$ so that $|\Phi(A^*) - \Phi(A_\epsilon)| \leq \epsilon$.

\section{Experiments on Rectangles}

We implemented 5 rectangle scanning algorithms. For baselines, we consider (1) Scanning all rectangles without sampling (based on common software for disks [13]) (SatScan (no sampling)), (2) Scanning all rectangles on one random sample [1] (SatScan), and (3) Scanning all rectangles on two random samples $N$ and $S$ [19] (netScan). Then we compare our algorithms which first round to a grid then (4) Efficiently enumerate the grid rectangles (gridScan, Theorem 3), or (5) Evaluate the maximum grid rectangle in $O(r^3)$ time [5] for a linear $\phi$ (gridScan\_linear, Section 4.1) and using the linearization for non-linear $\phi$ (Section 5). This is the core operation within spatial scan statistics; it is typically run 1000 times to detect a region and determine significance [12], therefore scalability of this operation is paramount. Solutions with approximate $\phi$ within $\epsilon$-error retain high statistical power [19], so it will be useful to directly compare the runtime performance of these algorithms which allow approximation.
Computing Approximate Statistical Discrepancy

Table 2 Runtimes on 1000 points with 1% error, over 20 trials; roughly \( n = 19 \) and \( s = 350 \).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SatScan (no sampling)</td>
<td>5287</td>
</tr>
<tr>
<td>SatScan</td>
<td>7.44</td>
</tr>
<tr>
<td>netScan</td>
<td>0.0279</td>
</tr>
<tr>
<td>gridScan</td>
<td>0.0194</td>
</tr>
<tr>
<td>gridScan_linear</td>
<td>0.0082</td>
</tr>
</tbody>
</table>

Figure 4 Trend of time versus error for on linear (left) and non-linear (right) functions.

First, fixing a tolerable error at 1% of \( \phi(A^*) \), we run each algorithm on \( m = 1000 \) points, for a planted range with 5% of the data, and use \( \phi \) as the Kuldorff scan statistic [12]. The results are in Table 2. All sampling methods drastically improve over the brute force approach, and using two-level sampling significantly improves over one random sample. Our method (gridScan_linear) improves over the previous best (netScan) by a factor of about 3.5.

We also compare the time-accuracy trade-off for sampling-based algorithms on \( m = 1 \) million points. SatScan without sampling is not tractable at this scale, so is not compared. We again plant a random rectangle \( A \) overlapping 1% of the data. Within \( A \), points are made red (measured value 1) at rate 0.08, and outside at rate 0.01. The runtime includes the time to construct the grid, but not time to generate the initial sample – common to all algorithms. We calculate \( \Phi(A^*) - \Phi(\hat{A}) \) for the planted \( A^* \) and found \( \hat{A} \) regions, using a linear \( \phi(m,b) = \frac{1}{\sqrt{2}}(m-b) \) function and the non-linear Kuldorff [12] \( \phi \) function. Figure 4 shows a kernel regression trend line (with 1 std-dev error bars) for 300 trials with various \( n,s \) values, always maintaining \( n \approx \sqrt{s} \) as suggested the sampling theorems. Again gridScan_linear is much faster than gridScan, which is slightly faster than netScan, which is significantly faster than SatScan. The improvement is more pronounced in the non-linear setting where \( \phi \) is steeper; this is perhaps surprisingly even true for gridScan_linear which has an extra \( \sqrt{1/\varepsilon} \)-factor in runtime in that case due to the multiple linear functions considered.

Ultimately, these plots show that discrete geometric approaches providing asymptotically efficient algorithms also give significant empirical improvements, even compared to the ubiquitous and simple random sampling approaches.

References

Diversity Maximization in Doubling Metrics

Alfonso Cevallos
Swiss Federal Institute of Technology (ETH), Switzerland
alfonso.cevallos@ifor.math.ethz.ch
https://orcid.org/0000-0001-8622-5830

Friedrich Eisenbrand1
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
friedrich.eisenbrand@epfl.ch
https://orcid.org/0000-0001-7928-1076

Sarah Morell2
Technische Universität Berlin (TU Berlin), Germany
morell@math.tu-berlin.de

Abstract
Diversity maximization is an important geometric optimization problem with many applications in recommender systems, machine learning or search engines among others. A typical diversification problem is as follows: Given a finite metric space \((X,d)\) and a parameter \(k \in \mathbb{N}\), find a subset of \(k\) elements of \(X\) that has maximum diversity. There are many functions that measure diversity. One of the most popular measures, called \textit{remote-clique}, is the sum of the pairwise distances of the chosen elements. In this paper, we present novel results on three widely used diversity measures: Remote-clique, remote-star and remote-bipartition.

Our main result are polynomial time approximation schemes for these three diversification problems under the assumption that the metric space is doubling. This setting has been discussed in the recent literature. The existence of such a PTAS however was left open.

Our results also hold in the setting where the distances are raised to a fixed power \(q \geq 1\), giving rise to more variants of diversity functions, similar in spirit to the variations of clustering problems depending on the power applied to the pairwise distances. Finally, we provide a proof of NP-hardness for remote-clique with squared distances in doubling metric spaces.

2012 ACM Subject Classification Theory of computation → Facility location and clustering

Keywords and phrases Remote-clique, remote-star, remote-bipartition, doubling dimension, grid rounding, \(\varepsilon\)-nets, polynomial time approximation scheme, facility location, information retrieval

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.33

Related Version A full version of the paper is available at [8], https://arxiv.org/abs/1809.09521.

1 Introduction

A dispersion or diversity maximization problem is as follows: Given a ground set \(X\) and a natural number \(k \in \mathbb{N}\), find a subset \(S \subseteq X\) among those of cardinality \(k\) that maximizes a certain diversity function \(\text{div}(S)\).
Diversity Maximization in Doubling Metrics

While diversity maximization has been of interest in the algorithms and operations research community for some time already, see e.g. [11, 5, 25, 20], the problem received considerable attention in the recent literature regarding information retrieval, recommender systems, machine learning and data mining, see e.g. [28, 29, 23, 24, 1].

Distances used in these applications may be metric or non-metric. However, most popular distances either are metric or correspond to the $q$-th power of metric distances for some $q > 1$. The cosine distance for the input vectors, after scaling all vectors to unit length.

Suppose that $X \subseteq \mathbb{R}^d \setminus \{0\}$, for example, is a popular non-metric measure of dissimilarity for text documents [26], which can be interpreted as the squared Euclidean distance of the input vectors, after scaling all vectors to unit length.

In this paper we focus on three popular diversity functions over metric spaces, see e.g. [11, 5, 2, 21, 17, 7, 6, 4]. In particular, for a given $n$-point metric space $(X, d)$, a constant $q \in \mathbb{R}_{\geq 1}$ and a parameter $k \in \mathbb{Z}$ with $2 \leq k \leq n$, we consider the family of problems

$$\max_{T \subseteq X, |T| = k} \text{div}^q(T),$$

where $\text{div}^q(T)$ corresponds to one of the following three diversity functions:

- **Remote-clique**: $\text{cl}^q(T) := \sum_{\{u,v\} \in \binom{T}{2}} d^q(u,v) = \frac{1}{2} \sum_{u,v \in T} d^q(u,v).

- **Remote-star**: $\text{st}^q(T) := \min_{z \in T} \sum_{u,v \in T \setminus \{z\}} d^q(z,u).

- **Remote-bipartition**: $\text{bp}^q(T) := \min_{L \subseteq T, |L| = |T|/2} \sum_{L \subseteq T \setminus L, r \in T \setminus L} d^q(L,r).

Here, $d^q(u,v)$ is the $q$-th power of the distance between $u$ and $v$. In the literature, these problems have been mainly considered for $q = 1$ to which we refer as standard remote-clique, remote-star and remote-bipartition respectively.

In the present work, we present polynomial time approximation schemes for the generalized versions ($q \geq 1$) of the remote-clique, remote-star and remote-bipartition problems in the case where the metric space is doubling. The latter is a general and robust class of metric spaces that have low intrinsic dimension. We provide a proper definition in Section 2.

**Contributions of this paper**

Suppose that $(X, d)$ is a metric space of bounded doubling dimension $D$ and that the power $q \geq 1$ is fixed. In this setting, our main results are as follows:

i) We show that there exist polynomial time approximation schemes (PTAS) for the remote-clique, remote-star and remote-bipartition problems. In other words, for each $\varepsilon > 0$ and for each of the three diversity functions $\text{cl}^q(T)$, $\text{st}^q(T)$ and $\text{bp}^q(T)$, there exists a polynomial time algorithm that computes a $k$-subset of $X$ whose diversity is at least $(1 - \varepsilon)$ times the diversity of the optimal set. We prove this result by means of a single and very simple algorithm that identifies a cluster which is then rounded, while all points outside of the cluster have to be in the optimal solution.

ii) For the standard ($q = 1$) remote-clique problem we refine our generic algorithm into a fast PTAS that runs in time $O(n(k + \varepsilon^{-D})) + (\varepsilon^{-1} \log k)O(\varepsilon^{-D}) \cdot k$.

iii) For the remote-bipartition problem, our algorithm assumes access to a polynomial time oracle that, for any $k$-set $T$, returns the value of $\text{bp}^q(T)$. For $q = 1$, this corresponds to the metric min-bisection problem, known to be NP-hard and admitting a PTAS [16]. We generalize this last result and provide a PTAS for min-bisection over doubling metric spaces for any constant $q \geq 1$, thus validating our main result.
Table 1 Current best approximation ratios and hardness results for remote-clique, remote-star and remote-bipartition with a highlight on our results. The sign † indicated that the result assumes hardness of the planted-clique problem.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Distance class</th>
<th>Unbounded dimension</th>
<th>Fixed (doubling) dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Approx. Hardness</td>
<td>Approx. Hardness</td>
</tr>
<tr>
<td>clique, (q = 1)</td>
<td>Metric</td>
<td>(1/2) [20, 5]</td>
<td>(1/2 + \varepsilon) [6]</td>
</tr>
<tr>
<td></td>
<td>(\ell_1) and (\ell_2)</td>
<td>PTAS [9, 10]</td>
<td>NP-hard [9]</td>
</tr>
<tr>
<td>clique, (q = 2)</td>
<td>Euclidean</td>
<td>PTAS [9, 10]</td>
<td>NP-hard [9]</td>
</tr>
<tr>
<td>star, (q = 1)</td>
<td>Metric</td>
<td>(1/2) [11]</td>
<td>(1/2 + \varepsilon) [8]</td>
</tr>
<tr>
<td>bipartition, (q = 1)</td>
<td>Metric</td>
<td>(1/3) [11]</td>
<td>(1/2 + \varepsilon) [8]</td>
</tr>
<tr>
<td>3 problems, any const. (q \geq 1)</td>
<td>Metric</td>
<td>–</td>
<td>(2^{-q} + \varepsilon) [8]</td>
</tr>
</tbody>
</table>

iv) We provide the first NP-hardness proof for remote-clique in fixed doubling dimension. More precisely, we prove that the version of remote-clique with squared Euclidean distances in \(\mathbb{R}^3\) is NP-hard.

Related work

For the standard case \(q = 1\) and for general metrics, Chandra and Halldórsson [11] provided a thorough study of several diversity problems, including remote-clique, remote-star and remote-bipartition. They observed that all three problems are NP-hard by reductions from the CLIQUE-problem and provided a \(\frac{1}{2}\)-factor and a \(\frac{1}{3}\)-factor approximation algorithm for remote-star and remote-bipartition respectively. Several approximation algorithms are known for remote-clique as well [25, 20, 5] with the current best factor being \(\frac{1}{2}\).

Remark. Borodin et al. [6] proved that the approximation factor of \(\frac{1}{2}\) is best possible for standard remote-clique over general metrics under the assumption that the planted-clique problem [3] is hard. In the full version we prove that, under the same assumption and for any \(q \geq 1\), neither remote-clique, remote-star nor remote-bipartition admits a constant approximation factor higher than \(2^{-q}\). Thus, none of the three problems nor their generalizations for \(q \geq 1\) admits a PTAS over general metrics.

In terms of relevant special cases for standard remote-clique, Ravi et al. [25] provided an efficient exact algorithm for instances over the real line, and a factor of \(\frac{2}{\pi}\) over the Euclidean plane. Later on, Fekete and Meijer [15] provided the first PTAS for this problem for fixed-dimensional \(\ell_1\) distances, and an improved factor of \(\frac{2\sqrt{2}}{\pi}\) over the Euclidean plane. Very recently, Cevallos et al. [9, 10] provided PTASs over \(\ell_1\) and \(\ell_2\) distances of unbounded dimension as well as for distances of negative type, a class that contains some popular non-metric distances including the cosine distance. We remark however that the running times of all previously mentioned PTASs [15, 9, 10] have a dependence on \(n\) given by high-degree polynomials (in the worst case) and thus are not suited for large data sets.

For remote-star and remote-bipartition, to the best of the authors’ knowledge there were no previous results in the literature on improved approximability for any fixed-dimensional setting, nor for other non-trivial special settings beyond general metrics. Moreover, there was no proof of NP-hardness for any of the three problems in a fixed-dimensional setting. In particular, showing NP-hardness of a fixed-dimensional geometric version of remote-clique was left as an open problem in [15].
Further related results and implications

In applications of diversity maximization in the area of information retrieval, common challenges come from the fact that the data sets are very large and/or are naturally embedded in a high dimensional vector space. There is active research in dimensionality reduction techniques, see [13] for a survey. It has also been remarked that in many scenarios such as human motion data and face recognition, data points have a hidden intrinsic dimension that is very low and independent from the ambient dimension, and there are ongoing efforts to develop algorithms and data structures that exploit this fact, see [27, 22, 14, 18]. One of the most common and theoretically robust notions of intrinsic dimension is precisely the doubling dimension. We remark that our algorithm does not need to embed the input points into a vector space (of low dimension or otherwise) and does not require knowledge of the doubling dimension, as this parameter only plays a role in the run-time analysis.

A sensible approach when dealing with very large data sets is to perform a core-set reduction of the input as a pre-processing step. This procedure quickly filters through the input points and discards most of them, leaving only a small subset – the core-set – that is guaranteed to contain a near-optimal solution. There are several recent results on core-set reductions for standard ($q = 1$) dispersion problems, see [21, 2, 7]. In particular, Ceccarello et al. [7] recently presented a PTAS-preserving reduction (resulting in an arbitrarily small deterioration of the approximation factor) for all three problems in doubling metric spaces, with the existence of a PTAS left open. Their construction allows for our algorithm to run in a machine of restricted memory and adapts it to streaming and distributed models of computation. Besides showing that a PTAS exists, we can also combine our results with theirs. We refer the interested reader to the previously mentioned references and limit ourselves to remark a direct consequence of Theorem 4 and [7, Theorems 3 and 9].

▶ Corollary 1. For $q = 1$ and any constant $\varepsilon > 0$, our three diversity problems over metric spaces of constant doubling dimension $D$ admit $(1 - \varepsilon)$-approximations that execute as single-pass and 2-pass streaming algorithms, in space $O(\varepsilon^{-D}k^2)$ and $O(\varepsilon^{-D}k)$ respectively.

Organization of the paper. In Section 2, we provide some needed notation and background techniques. Section 3 presents our general algorithm (Theorem 4) and Section 4 is dedicated to the NP-hardness result (Theorem 8). Due to space constraints, the description of the faster PTAS for standard remote-clique and the PTAS for the generalized min-bisection problem as well as the proofs of some lemmas have been deferred to the full version of this paper [8].

2 Preliminaries

A (finite) metric space is a tuple $(X, d)$, where $X$ is a finite set and $d : X \times X \to \mathbb{R}_{\geq 0}$ is a symmetric distance function that satisfies the triangle inequality with $d(u, u) = 0$ for each point $u \in X$. For a point $u \in X$ and a parameter $r \in \mathbb{R}_{\geq 0}$, the ball centered at $u$ of radius $r$ is defined as $B(u, r) := \{v \in X : d(u, v) \leq r\}$. The doubling dimension of $(X, d)$ is the smallest $D \in \mathbb{R}_{\geq 0}$ such that any ball in $X$ can be covered by at most $2^D$ balls of half its radius. In other words, for each $u \in X$ and $r > 0$, there exist points $v_1, \ldots, v_t \in X$ with $t \leq 2^D$ such that $B(u, r) \subseteq \cup_{i=1}^t B(v_i, r/2)$. A family of metric spaces is doubling if their doubling dimensions are bounded by a constant. It is well known that all metric spaces induced by a normed vector space of bounded dimension are doubling.

We rely on the standard cell-decomposition technique and grid-rounding, see [19]. We assume without loss of generality that the diameter of $(X, d)$, i.e. the largest distance between two points, is 1. For a parameter $\delta > 0$, the following greedy procedure partitions $X$ into
cells of radius $\delta$. Initially, define all points in $X$ to be white. While there exist white points, pick one that we call $u$, color it red and assign all white points $v \in X$ with $d(u, v) \leq \delta$ to $u$ and color them blue. A cell is now comprised of a red point, declared to be the cell center, and all the blue points assigned to it. Grid rounding means to move or round each point to its respective cell center. This incurs a location error of at most $\delta$ for each point.

How many cells and thus different points does this algorithm produce? If $(X, d)$ is of constant doubling dimension $D$, a direct consequence of the definition of $D$ is that for any parameters $r$ and $\rho$ in $\mathbb{R}_{>0}$, a ball of radius $r$ can be covered by at most $(2/\rho)^D$ balls of radius $\rho r$. Since $X$ is contained in a ball of radius 1, the number of cells produced is bounded by $(4/\delta)^D$. Indeed, $X$ can be covered by $(4/\delta)^D$ balls of radius $\delta/2$ and each such ball contains at most one cell center since, by construction, the distance between any two cell centers is strictly larger than $\delta$. Notice that this procedure executes in time $O((\# \text{ cells}) \cdot |X|)$ and that it requires no knowledge of the value of the doubling dimension $D$.

The following two lemmas correspond respectively to standard inequalities used for powers of metric distances and to trivial relations among our three diversity functions, see also [12]. Their proofs are deferred to the full version.

**Lemma 2.** Fix a constant $q \geq 1$. For any three points $u, v, w \in X$ one has

$$d^q(u, w) \leq 2^{q-1} \left[ d^q(u, v) + d^q(v, w) \right] \quad \text{or equivalently} \quad d^q(u, v) \geq 2^{-(q-1)} d^q(u, w) - d^q(v, w).$$

(1)

For any numbers $x, y \in \mathbb{R}_{\geq 0}$ and $0 \leq \varepsilon \leq 1$,

$$(x + \varepsilon y)^q \leq x^q + 2^q \varepsilon \cdot \max\{x^q, y^q\}.$$  

(3)

**Lemma 3.** Fix a constant $q \geq 1$. For any $k$-set $T \subseteq X$,

$$\frac{k}{2} \cdot \text{st}^q(T) \leq \text{cl}^q(T) \leq 2^{q-1} k \cdot \text{st}^q(T) \quad \text{and}$$

$$\frac{2(k-1)}{k} \cdot \text{bp}^q(T) \leq \text{cl}^q(T) \leq (2^q + 1) \cdot \text{bp}^q(T) \quad \text{(assuming that $k$ is even).}$$

(4)

(5)

Whenever we deal with remote-bipartition, we assume for simplicity that $k$ is even – all our results can easily be extended to the odd case, up to a change in constants by a factor $2^{O(n)}$. Therefore, the diversity functions correspond to the sum of $\binom{k}{2}$, $(k-1)$ and $k^2/4$ terms, respectively for remote-clique, remote-star and remote-bipartition. Consequently, for each function $\text{div}^q$ and for a given instance, we fix an optimal $k$-set denoted by $\text{OPT}_{\text{div}^q}$ and define its average optimal value $\Delta_{\text{div}^q}$ as follows:

$$\Delta_{\text{cl}^q} := \text{cl}^q(\text{OPT}_{\text{cl}^q})/\binom{k}{2},$$

$$\Delta_{\text{st}^q} := \text{st}^q(\text{OPT}_{\text{st}^q})/(k-1),$$

$$\Delta_{\text{bp}^q} := \text{bp}^q(\text{OPT}_{\text{bp}^q})/(k^2/4).$$

Whenever the diversity function $\text{div}^q$ is clear from context, or for general statements on all three functions, we use $\text{OPT}$ and $\Delta$ as short-hands for $\text{OPT}_{\text{div}^q}$ and $\Delta_{\text{div}^q}$ respectively.

**Remark.** It directly follows from Lemma 3 that for a common metric space and common parameters $q \geq 1$ and $k$, the average optimal values $\Delta_{\text{cl}^q}$, $\Delta_{\text{st}^q}$ and $\Delta_{\text{bp}^q}$ are all just a constant away from each other (a constant $2^{O(n)}$ that is independent of $n$ and $k$). We heavily use this property linking our three problems in the proof of our key structural result (Theorem 5). A similar result does not extend to other common diversity maximization problems such as remote-edge, remote-tree and remote-cycle, see [11] for definitions. This seems to be a bottleneck for possibly adapting our approach to those problems.
3 A PTAS for all three diversity problems

We now come to our main result which is the following theorem.

**Theorem 4.** For any constant $q \in \mathbb{R}_{\geq 1}$, the $q$-th power versions of the remote-clique, remote-star and remote-bipartition problems admit PTASs over doubling metric spaces.

Let us fix a constant error parameter $\epsilon > 0$. Our algorithm is based on grid rounding. However, if we think about the case $q = 1$, a direct implementation of this technique requires a cell decomposition of radius $O(\epsilon \cdot \Delta)$, which is manageable only if $\Delta$ is large enough with respect to the diameter. Otherwise, the number of cells produced may be super-constant in $n$. Hence, a difficult instance is one where $\Delta$ is very small, which intuitively occurs only in the degenerate case where most of the input points are densely clustered in a small region, with very few points outside of it. The algorithmic idea is thus to partition the input points into a *main cluster* and a collection of *outliers*, and treat these sets differently.

### 3.1 Key structural result

We identify in any instance a main cluster containing most of the input points. This cluster corresponds to a ball with a radius that is bounded with respect to $\Delta^{1/q}$. Thanks to the nature of the diversity functions, we can guarantee that *all outliers are contained in OPT*.

**Theorem 5.** Fix a constant $q \geq 1$. For each diversity function $\text{div}^q$ in \{cl$q$, st$q$, bp$q$\} and a fixed optimal $k$-set $\text{OPT}_{\text{div}^q}$, there is a point $z_0 = z_0(\text{div}^q)$ in $\text{OPT}_{\text{div}^q}$ so that

$$X \setminus B(z_0, c_{\text{div}^q}(\Delta_{\text{div}^q})^{1/q}) \subseteq \text{OPT}_{\text{div}^q},$$

where $c_{\text{cl}^q} = 2$, $c_{\text{st}^q} = 4$, and $c_{\text{bp}^q} = 6$.

**Proof.** For each function $\text{div}^q$ in \{cl$q$, st$q$, bp$q$\}, let $z_0 = z_0(\text{div}^q)$ be the center of the minimum weight spanning star in $\text{OPT}_{\text{div}^q}$ so that $st^q(\text{OPT}_{\text{div}^q}) = \sum_{u \in \text{OPT}_{\text{div}^q}} d^q(z_0, u)$. Consider a point $s = s(\text{div}^q)$ outside of the ball $B(z_0, c_{\text{div}^q}(\Delta_{\text{div}^q})^{1/q})$, i.e.

$$d^q(z_0, s) > (c_{\text{div}^q})^q \cdot \Delta_{\text{div}^q}.$$  

(6)

Assume that $s$ is not in $\text{OPT}_{\text{div}^q}$ and define the $k$-set $\text{OPT}_{\text{div}^q}' := \text{OPT}_{\text{div}^q} \cup \{s\} \setminus \{z_0\}$. We will show that the optimality of $\text{OPT}_{\text{div}^q}$ is not contradicted by the optimality of $\text{OPT}_{\text{div}^q}'$. To simplify notation in the remainder of the proof, we make the corresponding function clear from context and remove the subscripts $\text{div}^q$.

For remote-clique, we have

$$\text{cl}^q(\text{OPT}') - \text{cl}^q(\text{OPT}) = \sum_{u \in \text{OPT} \setminus \{z_0\}} [d^q(s, u) - d^q(z_0, u)] \geq \sum_{u \in \text{OPT} \setminus \{z_0\}} [2^{-(q-1)}d^q(z_0, s) - 2d^q(z_0, u)] \quad \text{(by (2))}$$

$$= k - 2^{(q-1)}d^q(z_0, s) - 2 \cdot \text{st}^q(\text{OPT}) \quad \text{(by choice of } z_0)$$

$$> \frac{k - 2^{(q-1)}(2q \Delta)}{k} \cdot \text{cl}^q(\text{OPT}) \quad \text{(by (6) and (4))}$$

$$= 2(k - 1) \Delta - 2(k - 1) \Delta = 0 \quad \text{(by def. of } \Delta).$$
For remote-star, let \( z \) be the center of the minimum weight spanning star in \( \text{OPT}' \) so that \( \text{st}^q(\text{OPT}') = d^q(z, s) + \sum_{u \in \text{OPT}' \setminus \{z_0\}} d^q(z, u) \). We claim that
\[
d^q(z_0, z) \leq 2^q \Delta,
\]
as otherwise we obtain
\[
\text{st}^q(\text{OPT}) + \text{st}^q(\text{OPT}') = d^q(z, s) + \sum_{u \in \text{OPT'} \setminus \{z_0\}} [d^q(z_0, u) + d^q(z, u)] \\
\geq 2^{-(q-1)} \sum_{u \in \text{OPT'} \setminus \{z_0\}} d^q(z, z) \\
\geq \frac{k}{2^{q-1}} (2^q \Delta) = 2(k-1)\Delta = 2 \cdot \text{st}^q(\text{OPT}) \quad \text{(negating (7)).}
\]
Inequality (7) implies in particular that \( z \neq s \), hence \( z \in \text{OPT} \). Notice by the minimality of the remote-star function that \( \text{st}^q(\text{OPT}) \leq \sum_{u \in \text{OPT}} d^q(z, u) \). By inequalities (2), (6) and (7), we obtain
\[
\text{st}^q(\text{OPT}') - \text{st}^q(\text{OPT}) \geq \sum_{u \in \text{OPT'}} d^q(z, u) - \sum_{u \in \text{OPT}} d^q(z, u) = d^q(z, s) - d^q(z, z_0) \\
\geq 2^{-(q-1)} d^q(z_0, s) - 2d^q(z_0, z) \\
> 2^{-(q-1)} (4^q \Delta) - 2(2^q \Delta) = 0.
\]
For remote-bipartition, let \( \text{OPT}' = L' \cup R \) be the minimum weight bipartition of \( \text{OPT}' \) so that \( \text{bp}^q(\text{OPT}') = \sum_{\ell \in L', r \in R} d^q(\ell, r) \). Assume without loss of generality that \( s \in L' \). We claim that
\[
\sum_{r \in R} d^q(z_0, r) \leq \frac{2^q + 1}{2} k \Delta,
\]
as otherwise we obtain
\[
\text{bp}^q(\text{OPT}) \geq \frac{1}{2^q + 1} \text{cl}^q(\text{OPT}) \geq \frac{k}{2(2^q + 1)} \text{st}^q(\text{OPT}) \\
= \frac{k}{2(2^q + 1)} \sum_{u \in \text{OPT}} d^q(z_0, u) \geq \frac{k}{2(2^q + 1)} \sum_{r \in R} d^q(z_0, r) \quad \text{(as } R \subseteq \text{OPT}) \\
> \frac{k}{2(2^q + 1)} \frac{2^q + 1}{2} k \Delta = \frac{k^2}{4} \Delta = \text{bp}^q(\text{OPT}) \quad \text{(negating (8)).}
\]
Define \( L := L' \cup \{z_0\} \setminus \{s\} \) and notice that \( L \cup R = \text{OPT} \). By the minimality of the remote-bipartition function, \( \text{bp}^q(\text{OPT}) \leq \sum_{\ell \in L} \sum_{r \in R} d^q(\ell, r) \). Hence,
\[
\text{bp}^q(\text{OPT}') - \text{bp}^q(\text{OPT}) \geq \sum_{\ell \in L} \sum_{r \in R} d^q(\ell, r) - \sum_{\ell \in L} \sum_{r \in R} d^q(\ell, r) \\
= \sum_{r \in R} [d^q(s, r) - d^q(z_0, r)] \\
\geq \sum_{r \in R} [2^{-(q-1)} d^q(z_0, s) - 2d^q(z_0, r)] \quad \text{(by (2))}
\]
\[
> \frac{|R|}{2^{q-1}} (6^q \Delta) - 2 \sum_{r \in R} d^q(z_0, r) \quad \text{(by (6))} \\
\geq 3^q k \cdot \Delta - (2^q + 1) k \cdot \Delta \geq 0. \quad \text{(by (8)).}
\]
This completes the proof of the theorem.
33:8 Diversity Maximization in Doubling Metrics

3.2 The algorithm

For any diversity function and a fixed optimal k-set, we refer to the ball \( B := B(z_0, c\Delta^{1/q}) \) defined in Theorem 5 as the main cluster and to \( z_0 \) as the instance center. Our algorithm consists of two phases: Finding the main cluster \( B \) and performing grid rounding on \( B \). We remark that for a well-dispersed instance, \( B \) may well contain all input points. In that case, our algorithm amounts to a direct application of the grid rounding procedure.

Finding the main cluster

There are several possible ways to (approximately) find \( B \). For simplicity, we present a naive approach based on exhaustive search. A smarter technique is described in the full version, where we provide a more refined algorithm for standard remote-clique.

Assuming without loss of generality that the instance diameter is 1, we obtain for each diversity function the bounds \( 1/k^2 \leq \Delta^{1/q} \leq 1 \). Hence, by performing \( O(\log k) \) trials, we can “guess” the value of \( \Delta^{1/q} \) up to a constant factor arbitrarily close to one, which means that for any constant \( \lambda > 0 \), we can find an estimate \( \Delta' \) so that \( (1 - \lambda)\Delta^{1/q} \leq \Delta'^{1/q} \leq \Delta^{1/q} \). Similarly, by trying out all \( n \) input points, we can “guess” the instance center \( z_0 \). For each one of these guesses, we perform the second phase (described in the next paragraph) and output the best \( k \)-set found over all trials. To simplify our exposition, we assume in what follows that we have found \( \Delta^{1/q} \) and \( z_0 \) (and thus \( B \)) exactly. Our analysis can be adapted to any constant-factor estimation of \( \Delta^{1/q} \), as it is enough to find a slightly larger ball \( B' \) containing \( B \) and to slightly change the value of constant \( c \). More precisely, if we have an estimate \( \Delta' \) so that \( (1 - \lambda)\Delta^{1/q} \leq \Delta'^{1/q} \leq \Delta^{1/q} \) and we set \( c' := \frac{c}{1 - \lambda} \), then \( B' := B(z_0, c'\Delta^{1/q}) \) is guaranteed to contain \( B \) and hence all points outside of \( B' \) are in OPT.

Rounding the cluster

We now assume that we have found the main cluster \( B \) (see the previous paragraph). For a constant \( \delta > 0 \) to be defined later, with \( 1/\delta = \Theta(2^q/\varepsilon) \), we perform a cell decomposition of radius \( \delta\Delta^{1/q} \) over \( B \). As the radius of ball \( B \) is \( c\Delta^{1/q} \), this decomposition produces at most \( (4 \cdot \frac{\Delta^{1/q}}{2\Delta^{1/q}})^D = (4c/\delta)^D = O(2^q/\varepsilon)^D \) cells, i.e. constantly many cells. Let \( \pi : B \to B \) be the function that maps each point to its cell center. For notational convenience, we extend this into a function \( \pi : X \to X \) by applying the identity on \( X \setminus B =: \bar{B} \) (and thinking of each point in \( \bar{B} \) as the center of its own cell). Finally, for any set \( T \subseteq X \), we denote by \( \hat{\pi}(T) \) the multiset over set \( \pi(T) \) having multiplicities \( |\pi^{-1}(u) \cap T| \) for each \( u \in \pi(T) \).

Next, we perform exhaustive search to find a \( k \)-set \( T \) in \( X \) with the property that

\[
\text{div}^q(\hat{\pi}(T)) \geq \text{div}^q(\hat{\pi}(\text{OPT})).
\]

This can be done in polynomial time as follows: We try out all multisets in \( \hat{\pi}(X) \) that a) contain \( \bar{B} \) and b) have cardinality \( k \) counting multiplicities. Then, we keep the multiset with largest diversity and return any \( k \)-set \( T \) that is a pre-image of this multiset. Clearly, this search considers only \( k^{O(2^q/\varepsilon)^D} \) multisets and is bound to consider \( \hat{\pi}(\text{OPT}) \).

As mentioned in the introduction, our algorithm assumes access to a polynomial-time oracle that, for any \( k \)-set \( T \), returns the value of \( \text{div}^q(T) \) or a \((1 + \varepsilon)\)-factor estimate of it which is sufficient for our purposes. The use of this estimate produces a corresponding small deterioration in our final approximation guarantee, but for simplicity we ignore this in the remainder. No exact efficient algorithm is known to compute \( \text{bp}^q(T) \) for a given \( k \)-set \( T \). However, we provide a PTAS for this problem in the full version.
3.3 Analysis

What is the approximation guarantee of our algorithm? By an application of inequality (3), our cell decomposition gives the following guarantee for each pair of points.

**Lemma 6.** Let \( \pi : X \to X \) be a map such that \( d(u, \pi(u)) \leq \delta \Delta^{1/q} \) for each \( u \) in \( X \). Then, for any pair of points \( u, v \in X \),

\[
|d^q(u, v) - d^q(\pi(u), \pi(v))| \leq 2^{q+1} \delta \cdot \left( \Delta + \min\{d^q(u, v), d^q(\pi(u), \pi(v))\} \right).
\]

**Proof.** We consider two cases. If \( d(u, v) \leq d(\pi(u), \pi(v)) \), we have by hypothesis

\[
d^q(\pi(u), \pi(v)) \leq d(\pi(u), u) + d(u, v) + d(v, \pi(v))] \leq d(u, v) + 20 \Delta^{1/q} \]

\[
\leq d^q(u, v) + 2^{q+1} \delta \cdot \max\{\Delta, d^q(u, v)\} \leq d^q(u, v) + 2^{q+1} \delta \cdot \left( \Delta + d^q(u, v) \right),
\]

where we used inequality (3) in the second line. This proves the claim.

Similarly, if \( d(\pi(u), \pi(v)) < d(u, v) \), then

\[
d^q(u, v) \leq d^q(\pi(u), \pi(v)) + 2^{q+1} \delta \cdot \left( \Delta + d^q(\pi(u), \pi(v)) \right),
\]

which again proves the claim. ▶

Lemma 6, together with the definition of \( \Delta \), implies the following result whose proof is deferred to the full version.

**Lemma 7.** Let \( \pi : X \to X \) be a map such that \( d(u, \pi(u)) \leq \delta \Delta^{1/q} \) for each \( u \) in \( X \). Then, for each one of our three diversity functions and for each \( k \)-set \( T \subseteq X \),

\[
|\text{div}^q(T) - \text{div}^q(\hat{T}(T))| \leq 2^{q+1} \delta \cdot [\text{div}^q(\text{OPT}) + \text{div}^q(T)] \leq 2^{q+2} \delta \cdot \text{div}^q(\text{OPT}).
\]

Applying the previous lemma twice as well as inequality (9) once, we conclude that

\[
\text{div}^q(T) \geq \text{div}^q(\hat{T}(T)) - 2^{q+2} \delta \cdot \text{div}^q(\text{OPT}) \geq \text{div}^q(\hat{T}(\text{OPT})) - 2^{q+2} \delta \cdot \text{div}^q(\text{OPT})
\]

\[
\geq \text{div}^q(\text{OPT}) - 2^{q+3} \delta \cdot \text{div}^q(\text{OPT}) = (1 - 2^{q+3} \delta) \cdot \text{div}^q(\text{OPT}).
\]

Hence, in order to achieve an approximation factor of \( 1 - \varepsilon \), it suffices to select \( \delta := \varepsilon/2^{q+3} \).

The number of cells produced by the cell decomposition is thus bounded by \( (2^{q+5}e/\varepsilon)^D = O(2^q/\varepsilon)^D \). This completes the analysis of our algorithm and the proof of Theorem 4.

4 Proof of NP-hardness

In this section, we present the first proof of NP-hardness for any of the three diversity problems in fixed dimension (in fact, the only other diversity maximization problem known to be NP-hard in a fixed-dimensional setting is remote-edge [30]). In particular, we prove NP-hardness for the squared distances \((q = 2)\) version of remote-clique in the case where all input points are unit vectors in the Euclidean space \( \mathbb{R}^3 \), i.e. \( X \subseteq \mathbb{S}^2 \).

**Theorem 8.** The squared distances version \((q = 2)\) of the remote-clique problem is NP-hard over the three-dimensional Euclidean space.

We remark that squared Euclidean distances over unit vectors correspond precisely to the popular cosine distances, hence the case considered is highly relevant.

For a \( k \)-set \( T \subseteq \mathbb{S}^2 \) with Euclidean distances, the function \( c^2(T) := \sum_{(u,v) \in T} d^2(u,v) \) has very particular geometric properties related to the concept of centroid. The centroid of a
k-set $T$ is defined as $z_T := \frac{1}{k} \sum_{u \in T} u$. It represents the coordinate-wise average of the points in $T$. The following result greatly simplifies the computation of function $c^2(T)$ in terms of the centroid. We state it for a general dimension $D$ even though we only use it for the case $D = 3$. Its proof is deferred to the full version.

**Lemma 9.** For a $k$-set $T \subseteq S^{D-1} \subseteq \mathbb{R}^D$ with centroid $z_T := \frac{1}{k} \sum_{u \in T} u$, $$c^2(T) = k^2 \cdot (1 - \|z_T\|^2).$$

We present a reduction from the $K$-SUM problem which is known to be NP-hard: Given a set $M$ of integer numbers in the range $[-t, t]$ for some threshold $t$ and a positive integer $K$, determine whether there is a $K$-set $S \subseteq M$ that sums to zero. Given such an instance of $K$-SUM, we define the following instance $X \subseteq S^2$ of remote-clique with $q = 2$, $|X| = 2|M|$ and $k = 2K$, see Figure 1. For each $m \in M$, set $m' := \frac{m}{t \sqrt{K}}$ and define $$X := \left\{ \ell_m := (-\sqrt{1 - m'^2}, m', 0)^\top : m \in M \right\} \cup \left\{ r_m := (\sqrt{1 - m'^2}, 0, m')^\top : m \in M \right\}.$$ Due to the scaling down by a factor of $\frac{1}{t \sqrt{K}}$, the $y$- and $z$-components of all points in $X$ are upper bounded by $\frac{1}{K}$ in absolute value, while their $x$-components are lower bounded by $\sqrt{1 - \frac{1}{K}}$ in absolute value. The points are thus tightly clustered around one of the two antipodal points $\pm(1, 0, 0)$, and $X$ is partitioned into a left cluster and a right cluster.

From Lemma 9, it is clear that solving this instance of remote-clique is equivalent to finding the $k$-set whose centroid is closest to the origin. Hence, the proof of Theorem 8 is complete once we show the following claim.

**Lemma 10.** If $M$ has a $K$-set $S$ with zero sum, then $X$ has a $k$-set $T$ with centroid $z_T = 0$. Otherwise, for every $k$-set $T \subseteq X$ we have $\|z_T\| \geq \frac{1}{2K^3/2}$.

**Proof.** Suppose that $M$ has a $K$-set $S$ with zero sum and define the $k$-set $T := \{\ell_m, r_m : m \in S\} \subseteq X$. Recall that its centroid $z_T$ corresponds to the component-wise average of the points in $T$, so we analyze these components separately. In $z$, all points of $T$ on the left cluster are zero and those on the right cluster have a zero sum, so $(z_T)_z = 0$. In $y$, all points...
of $T$ on the right cluster are zero and those on the left cluster have a zero sum, so $(z_T)_y = 0$. And in $x$, each point $\ell_m$ of $T$ on the left cluster is canceled out by its paired point $r_m$ on the right cluster, so $(z_T)_x = 0$. Therefore, $z_T = 0$.

Finally, we prove the contrapositive of the second statement, i.e. we assume that there is a $k$-set $T \subseteq X$ with $\|z_T\| < \frac{1}{2K^{3/2}}$. The set $T$ must contain exactly $K$ points in the left cluster and $K$ points in the right cluster. Indeed, if $T$ had at most $K - 1$ points in the left cluster, then the $x$-component of its centroid would give

$$(z_T)_x \geq (K - 1)(-1) + (K + 1) \sqrt{1 - \frac{1}{K}} \geq -(K - 1) + (K + 1) \left(1 - \frac{1}{K}\right) = 1 - \frac{1}{K},$$

and hence $\|z_T\| \geq |(z_T)_x| \geq 1 - \frac{1}{K} > \frac{1}{2K^{3/2}}$ for $K \geq 2$ and $t \geq 1$, leading to a contradiction.

Let $T = L \cup R$ be the corresponding (balanced) bipartition of $T$ given by the left and right clusters. Each of $L$ and $R$ must correspond to a $K$-set of $M$ with zero sum. Otherwise, without loss of generality $L$ corresponds to a $K$-set $S$ of $M$ with sum at least 1, but then

$$(z_T)_y = \frac{1}{2K} \sum_{m \in S} m' = \frac{1}{2K^{3/2}} \sum_{m \in S} m \geq \frac{1}{2K^{3/2}}$$

and thus $\|z_T\| \geq |(z_T)_y| \geq \frac{1}{2K^{3/2}}$, again a contradiction. This completes the proof. ◀

References


33:12 Diversity Maximization in Doubling Metrics


On Polynomial Time Constructions of Minimum Height Decision Tree

Nader H. Bshouty
Department of Computer Science, Technion, Haifa, Israel
bshouty@cs.technion.ac.il

Waseem Makhoul
Department of Computer Science, Technion, Haifa, Israel
waseemmakhoul@gmail.com

Abstract
A decision tree $T$ in $B_m := \{0,1\}^m$ is a binary tree where each of its internal nodes is labeled with an integer in $[m] = \{1,2,\ldots,m\}$, each leaf is labeled with an assignment $a \in B_m$ and each internal node has two outgoing edges that are labeled with 0 and 1, respectively. Let $A \subset \{0,1\}^m$. We say that $T$ is a decision tree for $A$ if (1) For every $a \in A$ there is one leaf of $T$ that is labeled with $a$. (2) For every path from the root to a leaf with internal nodes labeled with $i_1, i_2, \ldots, i_k \in [m]$, a leaf labeled with $a \in A$ and edges labeled with $\xi_{i_1}, \ldots, \xi_{i_k} \in \{0,1\}$, $a$ is the only element in $A$ that satisfies $a_{i_j} = \xi_{i_j}$ for all $j = 1, \ldots, k$.

Our goal is to write a polynomial time (in $n := |A|$ and $m$) algorithm that for an input $A \subseteq B_m$ outputs a decision tree for $A$ of minimum depth. This problem has many applications that include, to name a few, computer vision, group testing, exact learning from membership queries and game theory.

Arkin et al. and Moshkov [4, 15] gave a polynomial time $(\ln |A|)$-approximation algorithm (for the depth). The result of Dinur and Steurer [7] for set cover implies that this problem cannot be approximated with ratio $(1 - o(1)) \cdot \ln |A|$, unless P=NP. Moshkov studied in [15, 13, 14] the combinatorial measure of extended teaching dimension of $A$, $ETD(A)$. He showed that $ETD(A)$ is a lower bound for the depth of the decision tree for $A$ and then gave an exponential time $ETD(A)/\log(ETD(A))$-approximation algorithm and a polynomial time $2(\ln 2)ETD(A)$-approximation algorithm.

In this paper we further study the $ETD(A)$ measure and a new combinatorial measure, $DEN(A)$, that we call the density of the set $A$. We show that $DEN(A) \leq ETD(A) + 1$. We then give two results. The first result is that the lower bound $ETD(A)$ of Moshkov for the depth of the decision tree for $A$ is greater than the bounds that are obtained by the classical technique used in the literature. The second result is a polynomial time $(\ln 2)DEN(A)$-approximation (and therefore $(\ln 2)ETD(A)$-approximation) algorithm for the depth of the decision tree of $A$.

We then apply the above results to learning the class of disjunctions of predicates from membership queries [5]. We show that the $ETD$ of this class is bounded from above by the degree $d$ of its Hasse diagram. We then show that Moshkov algorithm can be run in polynomial time and is $(d/\log d)$-approximation algorithm. This gives optimal algorithms when the degree is constant. For example, learning axis parallel rays over constant dimension space.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization

Keywords and phrases Decision Tree, Minimal Depth, Approximation algorithms

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.34

Related Version A full version of the paper is available at [6], https://arxiv.org/abs/1802.00233.
1 Introduction

Consider the following problem: Given an $n$-element set $A \subseteq B_m := \{0, 1\}^m$ from some class of sets $\mathcal{A}$ and a hidden element $a \in A$. Given an oracle that answers queries of the type: “What is the value of $a_i$?”. Find a polynomial time algorithm that with an input $A$, asks minimum number of queries to the oracle and finds the hidden element $a$. This is equivalent to constructing a minimum height decision tree for $A$. A decision tree is a binary tree where each internal node is labeled with an index from $[m]$ and each leaf is labeled with an assignment $a \in B_m$. Each internal node has two outgoing edges one that is labeled with 0 and the other is labeled with 1. A node that is labeled with $i$ corresponds to the query “Is $a_i = 0$?”. An edge that is labeled with $\xi$ corresponds to the answer $\xi$. This decision tree is an algorithm in an obvious way and its height is the worst case complexity of the number of queries. A decision tree $T$ is said to be a decision tree for $A$ if the algorithm that corresponds to $T$ predicts correctly the hidden assignment $a \in A$. Our goal is to construct a small height decision tree for $A \subseteq B_m$ in time polynomial in $m$ and $n := |A|$. We will denote by $\text{OPT}(A)$ the minimum height decision tree for $A$

This problem is related to the following problem in exact learning [1]: Given a class $C$ of boolean functions $f : X \rightarrow \{0, 1\}$. Construct in $\text{poly}(|C|, |X|)$ time an optimal adaptive algorithm that learns $C$ from membership queries. This learning problem is equivalent to constructing a minimum height decision tree for the set $A = \{a^{(i)}| a^{(i)}_j = f_i(x_j)\}$ where $f_i$ is the $i$th function in $C$ and $x_j$ is the $j$th instance in $X$. In computer vision the problem is related to minimizing the number of “probes” (queries) needed to determine which one of a finite set of geometric figures is present in an image [4]. In game theory the problem is related to the minimum number of turns required in order to win a guessing game.

1.1 Previous and New Results

In [4], Arkin et al. showed that (AMMRS-algorithm) if at every node the decision tree chooses $i$ that partitions the current set (the set of assignments that are consistent to the answers of the queries so far) as evenly as possible, then the height of the tree is within a factor of $\log |A|$ from optimal. I.e., $\log |A|$-approximation algorithm. Moshkov [15] analysis shows that this algorithm is $(\ln |A|)$-approximation algorithm. This algorithm runs in polynomial time in $m$ and $|A|$.

Hyafil and Rivest, [11], show that the problem of constructing a minimum depth decision tree is NP-Hard. They actually consider the average depth but their technique can be adopted to the minimum depth. The reduction of Laber and Nogueira, [12] to set cover with the inapproximability result of Dinur and Steurer [7] for set cover implies that it cannot be approximated to a factor of $(1 - o(1)) \cdot \ln |A|$ unless P=NP. Therefore, no better approximation ratio can be obtained if no constraint is added to the set $A$.

Moshkov, [13], studied the extended teaching dimension combinatorial measure, $\text{ETD}(A)$, of a set $A \subseteq B_m$. It is the maximum over all the possible assignments $b \in B_m$ of the minimum number of indices $I \subset [m]$ in which $b$ agrees with at most one $a \in A$. Moshkov showed two results. The first is that $\text{ETD}(A)$ is a lower bound for $\text{OPT}(A)$. The second is an exponential time algorithm that asks $(2\text{ETD}(A)/\log \text{ETD}(A)) \log n$ queries. This gives a $(\ln 2)(\ln |A|)/\log \text{ETD}(A)$-approximation (exponential time) algorithm (since $\text{OPT}(A) \geq \text{ETD}(A)$) and at the same time $2\text{ETD}(A)/\log \text{ETD}(A)$-approximation algorithm (since $\text{OPT}(A) \geq \log |A|$). Since many interesting classes have small ETD dimension, the latter result gives small approximation ratio but unfortunately Moshkov algorithm runs in exponential time. In [14], Moshkov gave a polynomial time $2(\ln 2)\text{ETD}(C)$-approximation algorithm.
In this paper we further study the ETD measure. We show that the above AMMRS-algorithm, [4], is polynomial time \((\ln 2)\text{ETD}(C)\)-approximation algorithm. This improves the \(2(\ln 2)\text{ETD}(C)\)-approximation algorithm of Moshkov.

Another reason for studying the ETD of classes is the following: If you find the ETD of the set \(A\) then you either get a lower bound that is better than the information theoretic lower bound \(\log |A|\) or you get an approximation algorithm with a better ratio than \(\ln |A|\). This is because if \(\text{ETD}(A) < \ln |A|\) then the AMMRS-algorithm has a ratio \((\ln 2)\text{ETD}(A)\) that is better than the \(\ln |A|\) ratio and if \(\text{ETD}(A) > \ln |A|\) then Moshkov lower bound, \(\text{ETD}(A)\), for \(\text{OPT}(A)\) is better than the information theoretic lower bound \(\log |A|\).

To get the above results, we define a new combinatorial measure called the density \(\text{DEN}(A)\) of the set \(A\). If \(Q = \text{DEN}(A)\) then there is a subset \(B \subseteq A\) such that an adversary can give answers to the queries that eliminate at most \(1/Q\) fraction of the number of elements in \(B\). This forces the learner to ask at least \(Q\) queries. We then show that \(\text{ETD}(A) \geq \text{DEN}(A) - 1\).

On the other hand, we show that if \(Q = \text{DEN}(A)\) then a query in the AMMRS-algorithm eliminates at least \((1 - 1/Q)\) fraction of the assignments in \(A\). This gives a polynomial time \((\ln 2)\text{DEN}(A)\)-approximation algorithm which is also a \((\ln 2)(\text{ETD}(A) + 1)\)-approximation algorithm.

In order to compare both algorithms we show that \((\text{ETD}(A) - 1)/\ln |A| \leq \text{DEN}(A) \leq \text{ETD}(A) + 1\) and for random uniform \(A\) (and therefore for almost all \(A\)), with high probability \(\text{DEN}(A) = \Theta(\text{ETD}(A)/\ln |A|)\). Since \(|A| > \text{ETD}(A)\), this shows that AMMRS-algorithm may get a better approximation ratio than Moshkov algorithm.

The inapproximability results follows from the reduction of Laber and Nogueira, [12] to set cover with the inapproximability result of Dinur and Steurer [7] and the fact that \(\text{DEN}(A) \leq \text{ETD}(A) + 1 \leq \text{OPT}(A) + 1\).

We then apply the above results to learning the class of disjunctions of predicates from a set of predicates \(F\) from membership queries [5]. We show that the ETD of this class is bounded from above by the degree \(d\) of its Hasse diagram. We then show that Moshkov algorithm, for this class, runs in polynomial time and is \((d/\log d)\)-approximation algorithm. Since \(|F| \geq d\) (and in many applications, \(|F| \gg d|\)), this improves the \(|F|\)-approximation algorithm SPEX in [5] when the size of Hasse diagram is polynomial. This also gives optimal algorithms when the degree \(d\) is constant. For example, learning axis parallel rays over constant dimension space.

\section{Definitions and Preliminary Results}

In this section we give some definitions and preliminary results.

\subsection{Notation}

Let \(B_m = \{0, 1\}^n\). Let \(A = \{a^{(1)}, \ldots, a^{(n)}\} \subseteq B_m\) be an \(n\)-element set. We will write \(|A|\) for the number of elements in \(A\). For \(h \in B_m\) we define \(A + h = \{a + h | a \in A\}\) where \(+\) (in the square brackets) is the bitwise exclusive or of elements in \(B_m\).

For integer \(q\) let \([q] = \{1, 2, \ldots, q\}\). Throughout the paper, \(\log x = \log_2 x\).

\subsection{Optimal Algorithm}

We denote by \(\text{OPT}(A)\) the minimum depth of a decision tree for \(A\). Our goal is to build a decision tree for \(A\) with small depth. Obviously

\[\log n \leq \text{OPT}(A) \leq n - 1\]

(1)

where \(n := |A|\). The following result is easy to prove (see the full paper [6])
Lemma 1. We have \( \text{OPT}(A) = \text{OPT}(A + h) \).

2.3 Extended Teaching Dimension

In this section we define the extended teaching dimension.

Let \( h \in B_m \) be any element. We say that a set \( S \subseteq [m] \) is a specifying set for \( h \) with respect to \( A \) if \( \{a \in A \mid (\forall i \in S)h_i = a_i\} \leq 1 \). That is, there is at most one element in \( A \) that is consistent with \( h \) on the entries of \( S \). Denote by \( \text{ETD}(A, h) \) the minimum size of a specifying set for \( h \) with respect to \( A \). The extended teaching dimension of \( A \) is

\[
\text{ETD}(A) = \max_{h \in B_m} \text{ETD}(A, h). \tag{2}
\]

We will write \( \text{ETD}_z(A) \) for \( \text{ETD}(A, 0) \). It is easy to see that

\[
\text{ETD}(A, h) = \text{ETD}_z(A + h) \text{ and } \text{ETD}(A) = \text{ETD}(A + h). \tag{3}
\]

We say that a set \( S \subseteq [m] \) is a strong specifying set for \( h \) with respect to \( A \) if either \( h \in A \) and \( \{a \in A \mid (\forall i \in S)h_i = a_i\} = 1 \), or \( \{a \in A \mid (\forall i \in S)h_i = a_i\} = 0 \). That is, if \( h \in A \) then there is exactly one element in \( A \) that is consistent with \( h \) on the entries of \( S \). Otherwise, no element in \( A \) is consistent with \( h \) on \( S \). Denote \( \text{SETD}(A, h) \) the minimum size of a strong specifying set for \( h \) with respect to \( A \). The strong extended teaching dimension of \( A \) is

\[
\text{SETD}(A) = \max_{h \in B_m} \text{SETD}(A, h). \tag{4}
\]

We will write \( \text{SETD}_z(A) \) for \( \text{SETD}(A, 0) \). It is easy to see that

\[
\text{SETD}(A, h) = \text{SETD}_z(A + h) \text{ and } \text{SETD}(A) = \text{SETD}(A + h). \tag{5}
\]

Obviously, \( \text{ETD}(A, h) \leq \min(m, n - 1) \) and \( \text{ETD}(A, h) \leq \text{SETD}(A, h) \leq \min(m, n) \).

We now show

Lemma 2. We have \( \text{ETD}(A, h) \leq \text{SETD}(A, h) \leq \text{ETD}(A, h) + 1 \) and therefore \( \text{ETD}(A) \leq \text{SETD}(A) \leq \text{ETD}(A) + 1 \).

Proof. The fact \( \text{ETD}(A, h) \leq \text{SETD}(A, h) \) follows from the definitions. Let \( S \subseteq [m] \) be a specifying set for \( h \) with respect to \( A \). Then for \( T := \{a \in A \mid (\forall i \in S)h_i = a_i\} \) we have \( t := |T| \leq 1 \). If \( t = 0 \) or \( h \in A \) then \( S \) is a strong specifying set for \( h \) with respect to \( A \). If \( t = 1 \) and \( h \notin A \) then for the element \( a \in T \) there is \( j \in [m] \) such that \( a_j \neq h_j \) and then \( S \cup \{j\} \) is a strong specifying set for \( h \) with respect to \( A \). This proves that \( \text{SETD}(A, h) \leq \text{ETD}(A, h) + 1 \).

The other claims follows immediately.

Obviously, for any \( B \subseteq A \)

\[
\text{ETD}(B) \leq \text{ETD}(A), \quad \text{SETD}(B) \leq \text{SETD}(A). \tag{6}
\]

2.4 Hitting Set

A hitting set for \( A \) is a set \( S \subseteq [m] \) such that for every non-zero element \( a \in A \) there is \( j \in S \) such that \( a_j = 1 \). That is, \( S \) hits every element in \( A \) except the zero element (if it exists). The size of the minimum size hitting set for \( A \) is denoted by \( \text{HS}(A) \).

We now show

Lemma 3. We have \( \text{HS}(A) = \text{SETD}_z(A) \). In particular, \( \text{SETD}(A, h) = \text{HS}(A + h) \) and \( \text{SETD}(A) = \max_{h \in B_m} \text{HS}(A + h) \).
Proof. If $0 \in A$ then \( \text{SETD}_z(A) \) is the minimum size of a set $S$ such that \( \{ a \in A \mid (\forall i \in S) a_i = 0 \} = \{0\} \) and if $0 \notin A$ then it is the minimum size of a set $S$ such that \( \{ a \in A \mid (\forall i \in S) a_i = 0 \} = \emptyset \). Therefore the set $S$ hits all the nonzero elements in $A$.

The other results follow from (5) and the definition of \( \text{SETD} \).

\[ \square \]

2.5 Density of a Set

In this section we define our new measure \( \text{DEN} \) of a set.

Let $A = \{ a^{(1)}, \ldots, a^{(n)} \} \subseteq B_m$. We define \( \text{MAJ}(A) \in B_m \) such that \( \text{MAJ}(A)_i = 1 \) if the number of ones in \( (a^{(1)}_i, \ldots, a^{(n)}_i) \) is greater or equal the number of zeros and \( \text{MAJ}(A)_i = 0 \) otherwise. We denote by \( \text{MAX}(A) \) the maximum number of ones in \( (a^{(1)}_i, \ldots, a^{(n)}_i) \) over all \( i = 1, \ldots, m \). Let

\[ \text{MAMI}(A) = \min_{h \in B_m} \text{MAX}(A + h) = \text{MAX}(A + \text{MAJ}(A)). \] (7)

For \( j \in [m] \) and \( \xi \in \{0, 1\} \) let \( A_{j,\xi} = \{ a \in A \mid a_j = \xi \} \). Then

\[ \text{MAMI}(A) = \max_j \min(|A_{j,0}|, |A_{j,1}|). \] (8)

We define the density of a set $A \subseteq B_m$ by

\[ \text{DEN}(A) = \max_{B \subseteq A} \frac{|B| - 1}{\text{MAMI}(B)}. \] (9)

Notice that since every \( j \in [m] \) can hit at most \( \text{MAX}(A) \) elements in $A$ we have

\[ \text{HS}(A) \geq \frac{|A| - 1}{\text{MAX}(A)}. \] (10)

3 Bounds for \( \text{OPT} \)

In this section we give upper and lower bounds for \( \text{OPT} \).

3.1 Lower Bound

Moshkov results in [13, 10] and the information theoretic bound in (1) give the following lower bound. We give the proof in the full paper [6] for completeness.

\[ \blacktriangleright \text{Lemma 4.} \quad [13, 10] \text{ Let } A \subseteq B_m \text{ be any set. Then } \text{OPT}(A) \geq \max(\text{ETD}(A), \log |A|). \]

Many lower bounds in the literature for \( \text{OPT}(A) \) are based on finding a subset $B \subseteq A$ such that for each query there is an answer that eliminates at most small fraction $E$ of $B$. Then \( (|B| - 1)/E \) is a lower bound for \( \text{OPT}(A) \). The best possible bound that one can get using this technique is exactly \( \text{DEN}(A) \) (Lemma 5), the density defined in Section 2.5. Lemma 6 shows that the lower bound \( \text{ETD}(A) \) for \( \text{OPT}(A) \) exceeds any such bound.

In the full paper [6] we prove

\[ \blacktriangleright \text{Lemma 5.} \quad \text{We have } \text{OPT}(A) \geq \text{DEN}(A). \]

\[ \blacktriangleright \text{Lemma 6.} \quad \text{We have } \text{ETD}(A) \geq \text{DEN}(A) - 1. \]
Minimal Height Decision Tree

**Proof.** By (7) and (9) there is \( B \subseteq A \) such that

\[
\text{DEN}(A) = \frac{|B| - 1}{\text{MAMI}(B)} = \frac{|B| - 1}{\text{MAX}(B + h)}
\]

where \( h = \text{MAJ}(B) \). Then

\[
\text{ETD}(A) \overset{(6)}{\geq} \text{ETD}(B) \overset{(2)}{\geq} \text{ETD}(B, h) \overset{L^2}{\geq} \text{SETD}(B, h) - 1 \overset{L^3}{=} \text{HS}(B + h) - 1
\]

\[
\overset{(10)}{\geq} \frac{|B| - 1}{\text{MAX}(B + h)} - 1 \overset{(11)}{=} \text{DEN}(A) - 1.
\]

In the full paper [6] we also prove

**Lemma 7.** We have \( \text{ETD}(A) \leq \ln |A| \cdot \text{DEN}(A) + 1 \).

It is also easy to see (by standard analysis using Chernoff Bound) that for a random uniform \( A \), with positive probability, \( \text{DEN}(A) = O(1) \) and \( \text{ETD}(A) = \Theta(\log |A|) \). See the proof sketch in the full paper [6]. So the bound in Lemma 7 is asymptotically best possible.

### 3.2 Upper Bounds

Moshkov [13, 10] proved the following upper bound. We gave the proof in the full paper [6] for completeness.

**Lemma 8.** [13, 10] Let \( A \subseteq \{0, 1\}^m \) of size \( n \). Then

\[
\text{OPT}(A) \leq \text{ETD}(A) + \frac{\text{ETD}(A)}{\log \text{ETD}(A)} \log n \leq \frac{2 \cdot \text{ETD}(A)}{\log \text{ETD}(A)} \log n.
\]

In [13, 10], Moshkov gave an example of a \( n \)-set \( A_E \subseteq \{0, 1\}^m \) with \( \text{ETD}(A_E) = E \) and \( \text{OPT}(A_E) = \Omega((E/\log E) \log n) \). So the upper bound in the above lemma is the best possible.

### 4 Polynomial Time Approximation Algorithm

Given a set \( A \subseteq B_m \). Can one construct an algorithm that finds a hidden \( a \in A \) with \( \text{OPT}(A) \) queries? Obviously, with unlimited computational power this can be done so the question is: How close to \( \text{OPT}(A) \) can one get when polynomial time \( \text{poly}(m, n) \) is allowed for the construction?

An exponential time algorithm follows from the following

\[
\text{OPT}(A) = \min_{i \in [m]} \max(\text{OPT}(A_{i,0}), \text{OPT}(A_{i,1}))
\]

where \( A_{i,\xi} = \{a \in A \mid a_i = \xi\} \). This algorithm runs in time at least \( m! \geq (m/e)^m \). See also [8, 3].

Can one give a better exponential time algorithm? In what follows (Theorem 9) we use Moshkov [13, 10] result (Lemma 8) to give a better exponential time approximation algorithm. In the full paper [6] we give another simple proof of the Moshkov [13, 10] result that in practice uses less number of specifying sets. When the extended teaching dimension is constant, the algorithm is \( O(1) \)-approximation algorithm and runs in polynomial time.
Theorem 9. Let $A$ be a class of sets $A \subseteq B_m$ of size $n$. If there is an algorithm that for any $h \in B_m$ and any $A \in A$ gives a specifying set for $h$ with respect to $A$ of size at most $E$ in time $T$ then there is an algorithm that for any $A \in A$ constructs a decision tree for $A$ of depth at most $E + \frac{E}{\log E} \log n \leq E + \frac{E}{\log E} \text{OPT}(A)$ queries and runs in time $O(T \log n + nm)$.

Proof. Follows immediately from Moshkov algorithm [13, 10]. See the full paper [6]. ◀

The following result immediately follows from Theorem 9.

Theorem 10. Let $A \subseteq B_m$ be a $n$-set. There is an algorithm that finds the hidden column in time $\left(\frac{m}{\text{ETD}(A)}\right) \cdot \text{ETD}(A) \cdot n \log n$ and asks at most $\frac{2 \cdot \text{ETD}(A) \cdot \log n}{\log \text{ETD}(A)} \leq \frac{2 \cdot \min(\text{ETD}(A), \log n)}{\log \text{ETD}(A)} \text{OPT}(A)$ queries.

In particular, if $\text{ETD}(A)$ is constant then the algorithm is $O(1)$-approximation algorithm that runs in polynomial time.

Proof. To find a specifying set for $h$ with respect to $A$ we exhaustively check each ETD($A$) row of $A$. Each check takes time $n$. Since the algorithm asks at most $\text{ETD}(A) \cdot \log n$ queries, the time complexity is as stated in the Theorem. ◀

Can one do it in $poly(m,n)$ time? Hyafil and Rivest, [11], show that the problem of finding $\text{OPT}$ is NP-Complete. The reduction of Laber and Nogueira, [12], of set cover to this problem with the inapproximability result of Dinur and Steurer [7] for set cover implies that it cannot be approximated to $(1 - o(1)) \cdot \ln n$ unless $P=NP$.

In [4], Arkin et al. showed that (the AMMRS-algorithm) if at the $i$th query the algorithm chooses an index $j$ that partitions the current node set (the elements in $A$ that are consistent with the answers until this node) $A$ as evenly as possible, that is, that maximizes $\min(|\{a \in A|a_j = 0\}|, |\{a \in A|a_j = 1\}|)$, then the query complexity is within a factor of $[\log n]$ from optimal. The AMMRS-algorithm, [4], runs in time $poly(m,n)$. Moshkov [4, 15] analysis shows that this algorithm is in $n$-approximation algorithm and therefore is optimal. In this section we will give a simple proof.

In [13, 10], Moshkov gave a simple ETD($A$)-approximation algorithm (Algorithm MEMB-HALVING-1 in [10]). He then gave another algorithm that achieves the query complexity in Lemma 8 (Algorithm MEMB-HALVING-2 in [10]). This is within a factor of $\frac{2 \cdot \min(\text{ETD}(A), \log n)}{\log \text{ETD}(A)}$ from optimal. This is better than the ratio $\ln n$, but, unfortunately, both algorithms require finding a minimum size specifying set and the problem of finding a minimum size specifying set for $h$ is NP-Hard, [16, 2, 9]. Moshkov gave in [14] a polynomial time $2(\ln 2)$-approximation algorithm.
Can one achieve a better approximation ratio? In the following we give a surprising result. We show that the AMMRS-algorithm asks \( \text{DEN}(A) \ln |A| \) queries. Therefore, it is a \((\ln 2)\text{DEN}(A)\)-approximation algorithm and therefore it is a \((\ln 2)\text{ETD}(A)\)-approximation algorithm. This also prove that it is a \(\ln |A|\)-approximation algorithm. We also show that no algorithm with query complexity \( (1 - \epsilon)\text{DEN}(A) \ln |A| \) is possible unless \( P=NP \).

**Theorem 11.** The AMMRS-algorithm runs in time \( O(mn) \) and finds the hidden element \( a \in A \) with at most
\[
\text{DEN}(A) \cdot \ln(n) \leq \min(\ln 2)\text{DEN}(A), \ln n) \cdot \text{OPT}(A) \\
\leq \min(\ln 2)(\text{ETD}(A) + 1), \ln n) \cdot \text{OPT}(A)
\]
queries.

**Proof.** Let \( B \) be any subset of \( A \). Then,
\[
\text{DEN}(B) \geq \frac{|B| - 1}{\text{MAMI}(B)}
\]
and therefore
\[
\text{MAMI}(B) \geq \frac{|B| - 1}{\text{DEN}(B)} \geq \frac{|B| - 1}{\text{DEN}(A)}.
\]

Since the AMMRS-algorithm chooses at each node in the decision tree the index \( j \) that maximizes \( \min(|B_{j,0}|, |B_{j,1}|) \) where \( B_{j,\xi} = \{ a \in B | a_j = \xi \} \) and \( B \) is the set of elements in \( A \) that are consistent with the answers until this node, we have
\[
\max(|B_{j,0}|, |B_{j,1}|) - 1 = |B| - 1 - \min(|B_{j,0}|, |B_{j,1}|) = (8) \frac{|B| - 1 - \text{MAMI}(B) \leq (|B| - 1) \left( 1 - \frac{1}{\text{DEN}(A)} \right)}.
\]
Therefore, for a node \( v \) of depth \( h \) in the decision tree, the set \( B(v) \) of elements in \( A \) that are consistent with the answers until this node contains at most
\[
(|A| - 1) \left( 1 - \frac{1}{\text{DEN}(A)} \right)^h + 1
\]
elements. Therefore the depth of the tree is at most \( \text{DEN}(A) \ln |A| \).

We now show that the query complexity of this algorithm is optimal unless \( P=NP \).

**Theorem 12.** Let \( \epsilon \) be any constant. There is no polynomial time algorithm that finds the hidden element with less than \((1 - \epsilon)\text{DEN}(A) \cdot \ln |A| \) unless \( P=NP \).

**Proof.** Suppose such an algorithm exists. Then \((1 - \epsilon)\text{DEN}(A) \ln |A| \leq (1 - \epsilon) \ln |A| \text{OPT}(A)\). That is, the algorithm is also \((1 - \epsilon)\ln |A|\)-approximation algorithm. Laber and Nogueira, [12] gave a polynomial time algorithm reduction of minimum depth decision tree to set cover and Dinur and Steurer [7] show that there is no polynomial time \((1 - o(1)) \cdot \ln |A| \) for set cover unless \( P=NP \). Therefore, such an algorithm implies \( P=NP \).
5 Applications to Disjunction of Predicates

In this section we apply the above results to learning the class of disjunctions of predicates from a set of predicates $\mathcal{F}$ from membership queries [5].

Let $C = \{f_1, \ldots, f_n\}$ be a set of boolean functions $f_i : X \rightarrow \{0,1\}$ where $X = \{x_1, \ldots, x_m\}$. Let $A_C = \{(f_i(x_1), \ldots, f_i(x_m)) \mid i = 1, \ldots, n\}$. We will write $\text{OPT}(A_C)$, $\text{ETD}(A_C)$, etc. as $\text{OPT}(C)$, $\text{ETD}(C)$, etc.

Let $\mathcal{F}$ be a set of boolean functions (predicates) over a domain $X$. We consider the class of functions $\mathcal{F}_\vee := \{\lor_{f \in \mathcal{F}} \mid S \subseteq \mathcal{F}\}$.

5.1 An Equivalence Relation Over $\mathcal{F}_\vee$

In this section, we present an equivalence relation over $\mathcal{F}_\vee$ and define the representatives of the equivalence classes. This enables us in later sections to focus on the representative elements from $\mathcal{F}_\vee$. Let $\mathcal{F}$ be a set of boolean functions over the domain $X$. The equivalence relation $=$ over $\mathcal{F}_\vee$ is defined as follows: two disjunctions $F_1, F_2 \in \mathcal{F}_\vee$ are equivalent ($F_1 = F_2$) if $F_1$ is logically equal to $F_2$. In other words, they represent the same function (from $X$ to $\{0,1\}$). We write $F_1 \equiv F_2$ to denote that $F_1$ and $F_2$ are identical; that is, they have the same representation. For example, consider $f_1, f_2 : \{0,1\} \rightarrow \{0,1\}$ where $f_1(x) = 1$ and $f_2(x) = x$. Then, $f_1 \lor f_2 = f_1$ but $f_1 \lor f_2 \neq f_1$.

We denote by $\mathcal{F}_\vee^*$ the set of equivalence classes of $=$ and write each equivalence class as $[F]$, where $F \in \mathcal{F}_\vee$. Notice that if $[F_1] = [F_2]$, then $[F_1 \lor F_2] = [F_1] = [F_2]$. Therefore, for every $[F]$, we can choose the representative element to be $G_F := \lor_{F' \in \mathcal{F}'} S$ where $S \subseteq \mathcal{F}$ is the maximum size set that satisfies $\lor S := \lor_{f \in S} F = F$. We denote by $G(\mathcal{F}_\vee)$ the set of all representative elements. Accordingly, $G(\mathcal{F}_\vee) = \{G_F \mid F \in \mathcal{F}_\vee\}$. As an example, consider the set $\mathcal{F}$ consisting of four functions $f_{11}, f_{12}, f_{21}, f_{22} : \{1,2\}^2 \rightarrow \{0,1\}$ where $f_{ij}(x_1, x_2) = [x_i \geq j]$ where $[x_i \geq j] = 1$ if $x_i \geq j$ and 0 otherwise. There are $2^4 = 16$ elements in $\text{Ray}_2^2 := \mathcal{F}_\vee$ and five representative functions in $G(\mathcal{F}_\vee)$: $G(\mathcal{F}_\vee) = \{f_{11} \lor f_{12} \lor f_{21} \lor f_{22}, f_{12} \lor f_{22}, f_{12}, f_{22}, 0\}$ (where 0 is the zero function).

5.2 A Partial Order Over $\mathcal{F}_\vee$ and Hasse Diagram

In this section, we define a partial order over $\mathcal{F}_\vee$ and present related definitions. The partial order, denoted by $\rightarrow$, is defined as follows: $F_1 \Rightarrow F_2$ if $F_1$ logically implies $F_2$. Consider the Hasse diagram $H(\mathcal{F}_\vee)$ of $G(\mathcal{F}_\vee)$ for this partial order. The maximum (top) element in the diagram is $G_{\max} := \lor_{f \in \mathcal{F}} f$. The minimum (bottom) element is $G_{\min} := \lor_{f \in \emptyset} f$, i.e., the zero function.

In a Hasse diagram, $G_1$ is a descendant (resp., ascendent) of $G_2$ if there is a (nonempty) downward path from $G_2$ to $G_1$ (resp., from $G_1$ to $G_2$), i.e., $G_1 \Rightarrow G_2$ (resp., $G_2 \Rightarrow G_1$) and $G_1 \neq G_2$. $G_1$ is an immediate descendant of $G_2$ in $H(\mathcal{F}_\vee)$ if $G_1 \Rightarrow G_2, G_1 \neq G_2$ and there is no $G \in G(\mathcal{F}_\vee)$ such that $G \neq G_1, G \neq G_2$ and $G_1 \Rightarrow G \Rightarrow G_2$. $G_1$ is an immediate ascendent of $G_2$ if $G_2$ is an immediate descendant of $G_1$.

We denote by $\text{De}(G)$ and $\text{As}(G)$ the sets of all the immediate descendants and immediate ascendants of $G$, respectively. The neighbours set of $G$ is $\text{Ne}(G) = \text{De}(G) \cup \text{As}(G)$. We further denote by $\text{DE}(G)$ and $\text{AS}(G)$ the sets of all $G$’s descendants and ascendants, respectively.

Definition 13. The degree of $G$ is $\text{deg}(G) = |\text{Ne}(G)|$ and the degree $\text{deg}(\mathcal{F}_\vee)$ of $\mathcal{F}_\vee$ is $\max_{G \in G(\mathcal{F}_\vee)} \text{deg}(G)$. 

ISAAC 2018
For $G_1$ and $G_2$, we define their lowest common ascendent (resp., greatest common descendant) $G = \text{lca}(G_1, G_2)$ (resp., $G = \text{gcd}(G_1, G_2)$) to be the minimum (resp., maximum) element in $\text{AS}(G_1) \cap \text{AS}(G_2)$ (resp., $\text{DE}(G_1) \cap \text{DE}(G_2)$).

The following result is from [5]

Lemma 14. Let $G_1, G_2 \in G(\mathcal{F}_\vee)$. Then, $\text{lca}(G_1, G_2) = G_1 \lor G_2$.

In particular, if $G_1, G_2$ are two distinct immediate descendants of $G$, then $G_1 \lor G_2 = G$.

5.3 Witnesses

In this subsection we define the term witness. Let $G_1$ and $G_2$ be elements in $G(\mathcal{F}_\vee)$. An element $a \in X$ is a witness for $G_1$ and $G_2$ if $G_1(a) \neq G_2(a)$.

For a class of boolean functions $C$ over a domain $X$ and a function $G \in C$ we say that a set of elements $W \subseteq X$ is a witness set for $G$ in $C$ if for every $G' \in C$ and $G' \neq G$ there is a witness in $W$ for $G$ and $G'$.

5.4 The Extended Teaching Dimension of $\mathcal{F}_\vee$

In this section we prove

Lemma 15. For every $h : X \rightarrow \{0,1\}$ if $h \not\equiv G_{\max}$ then $\text{ETD}(\mathcal{F}_\vee, h) = 1$. Otherwise, there is $G \in G(\mathcal{F}_\vee)$ such that

$$\text{ETD}(\mathcal{F}_\vee, h) \leq |\text{De}(G)| + \text{HS}(\text{As}(G) \land \bar{G}) \leq |\text{Ne}(G)| = \deg(G)$$

where $\text{As}(G) \land \bar{G} = \{s \land \bar{G} \mid s \in \text{As}(G)\}$. In particular,

$$\text{ETD}(\mathcal{F}_\vee) \leq \max_{G \in G(\mathcal{F}_\vee)} (|\text{De}(G)| + \text{HS}(\text{As}(G) \land \bar{G})) \leq \deg(\mathcal{F}_\vee).$$

Proof. Let $h : X \rightarrow \{0,1\}$ be any function. If $h \not\equiv G_{\max}$ then there is an assignment $a$ that satisfies $h(a) = 1$ and $G_{\max}(a) = 0$. Since for all $G \in G(\mathcal{F}_\vee)$, $G \not\equiv G_{\max}$ we have $G(a) = 0$. Therefore, the set $\{a\}$ is a specifying set for $h$ with respect to $\mathcal{F}_\vee$ and $\text{ETD}(\mathcal{F}_\vee, h) = 1$.

Let $h \equiv G_{\max}$. Consider any $G \in G(\mathcal{F}_\vee)$ such that $h \not\equiv G$ and for every immediate descendant $G'$ of $G$ we have $h \not\equiv G'$. Now for every immediate descendant $G''$ of $G$ find an assignment $a$ such that $G''(a) = 0$ and $h(a) = 1$. Then $a$ is a witness for $h$ and $G'$. Therefore, $a$ is also a witness for $h$ and every descendant of $G'$. Let $A$ be the set of all such assignments, i.e., for every descendant of $G$ one witness. Then $|A| \leq |\text{De}(G)|$ and $A$ is a witness set for $h$ and all the descendants of $G$. We note here that if $h = 0$ then $G = G_{\min}$ which has no immediate descendants and then $A = \emptyset$.

Consider a hitting set $B$ for $\text{As}(G) \land \bar{G}$ of size $\text{HS}(\text{As}(G) \land \bar{G})$. Now for every immediate ascendent $G''$ of $G$ find an assignment $b \in B$ such that $G''(b) \land \bar{G}(b) = 1$. Then $G''(b) = 1$ and $G(b) = 0$. Since $G(b) = 0$ we have $h(b) = 0$ and then $b$ is a witness for $h$ and $G''$. Therefore, $b$ is also a witness for $h$ and every ascendent of $G''$. Thus $B$ is a witness set for $h$ in all the ascendants of $G$.

Let $G_0$ be any element in $G(\mathcal{F}_\vee)$ (that is not a descendant or an ascendent). Consider $G_1 = \text{lca}(G, G_0)$. By Lemma 14, we have $G_1 = G \lor G_0$. Since $G_1$ is an ascendent of $G$ there is a witness $a \in B$ such that $G_1(a) = 1$ and $G(a) = 0$. Then $G_0(a) = 1$, $h(a) = 0$ and $a$ is a witness of $h$ and $G_0$. Therefore $A \cup B$ is a specifying set for $h$ with respect to $G(\mathcal{F}_\vee)$. Since for every $F \in \mathcal{F}_\vee$ we have $F = G_F \in G(\mathcal{F}_\vee)$, $A \cup B$ is also a specifying set for $h$ with respect to $\mathcal{F}_\vee$. 
Since
\[
\text{ETD}(\mathcal{F}_\forall, h) \leq |A| + |B| \leq |\text{De}(G)| + \text{HS}(\text{As}(\mathcal{G}) \land \mathcal{G})
\]
the result follows. ▶

In in the full paper [6] we show that
\[
\text{ETD}(\mathcal{F}_\forall) = \max_{G \in \mathcal{G}(\mathcal{F}_\forall)} \left( |\text{De}(G)| + \text{HS}(\text{As}(\mathcal{G}) \land \mathcal{G}) \right).
\]
We could have replaced $|\text{De}(G)|$ by $\text{HS}(\text{De}(G) \land \mathcal{G})$, but in the full paper [6] we show that
they are both equal.

The following result follows immediately from the proof of Lemma 15

$i)$ Lemma 16. For any $h : X \to \{0, 1\}$, a specifying set for $h$ with respect to $\mathcal{F}_\forall$ of size $\deg(\mathcal{F}_\forall)$ can be found in time $O(nm)$.

By Theorem 9 we have

$i)$ Theorem 17. There is an algorithm that learns $\mathcal{F}_\forall$ in time $O(nm)$ and asks at most
\[
\deg(\mathcal{F}_\forall) + \frac{\deg(\mathcal{F}_\forall)}{\log \deg(\mathcal{F}_\forall)} \log n \leq \left( \frac{\deg(\mathcal{F}_\forall)}{\log \deg(\mathcal{F}_\forall)} + 1 \right) \text{OPT}(\mathcal{F}_\forall)
\]

membership queries.

5.5 Learning Other Classes

If a specifying set of small size cannot be found in polynomial time then from Theorem 10, 11 and Lemma 15, we have

$i)$ Theorem 18. For a class $\mathcal{C}$ we have

1. There is an algorithm that learns $\mathcal{C}$ in time
\[
\binom{m}{\deg(\mathcal{C})} \cdot \text{ETD}(\mathcal{C}) \cdot n \log n
\]

and asks at most
\[
2 \cdot \frac{\text{ETD}(\mathcal{C}) \cdot \log n}{\log \text{ETD}(\mathcal{C})} \leq 2 \cdot \min\left(\frac{\text{ETD}(\mathcal{C}) \cdot \log n}{\log \text{ETD}(\mathcal{C})}\right) \text{OPT}(\mathcal{C})
\]

membership queries.

In particular, when $\text{ETD}(\mathcal{C})$ is constant the algorithm runs in polynomial time and its query complexity is (asymptotically) optimal.

2. There is an algorithm that learns $\mathcal{C}$ in time $O(nm)$ and asks at most
\[
\text{DEN}(\mathcal{C}) \cdot \log(n) \leq \min((\ln 2)\text{DEN}(\mathcal{C}), \ln n) \cdot \text{OPT}(\mathcal{C})
\]
\[
\leq \min((\ln (\text{ETD}(\mathcal{C}) + 1), \ln n) \cdot \text{OPT}(\mathcal{C})
\]

membership queries.
References


Improved Algorithms for the Shortest Vector Problem and the Closest Vector Problem in the Infinity Norm

Divesh Aggarwal\(^1\)  
Centre for Quantum Technologies and School of Computing, National University of Singapore  
dcsdiva@nus.edu.sg

Priyanka Mukhopadhyay\(^2\)  
Centre for Quantum Technologies, National University of Singapore  
mukhopadhyay.priyanka@gmail.com

Abstract

Ajtai, Kumar and Sivakumar [5] gave the first \(2^{O(n)}\) algorithm for solving the Shortest Vector Problem (SVP) on \(n\)-dimensional Euclidean lattices. The algorithm starts with \(N \in 2^{O(n)}\) randomly chosen vectors in the lattice and employs a sieving procedure to iteratively obtain shorter vectors in the lattice, and eventually obtaining the shortest non-zero vector. The running time of the sieving procedure is quadratic in \(N\). Subsequent works [7, 11] generalized the algorithm to other norms.

We study this problem for the special but important case of the \(\ell_\infty\) norm. We give a new sieving procedure that runs in time linear in \(N\), thereby improving the running time of the algorithm for SVP in the \(\ell_\infty\) norm. As in [6, 11], we also extend this algorithm to obtain significantly faster algorithms for approximate versions of the shortest vector problem and the closest vector problem (CVP) in the \(\ell_\infty\) norm.

We also show that the heuristic sieving algorithms of Nguyen and Vidick [23] and Wang et al. [27] can also be analyzed in the \(\ell_\infty\) norm. The main technical contribution in this part is to calculate the expected volume of intersection of a unit ball centred at origin and another ball of a different radius centred at a uniformly random point on the boundary of the unit ball. This might be of independent interest.

2012 ACM Subject Classification Theory of computation → Randomness, geometry and discrete structures

Keywords and phrases Lattice, Shortest Vector Problem, Closest Vector Problem, \(\ell_\infty\) norm

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.35

Related Version A full version of the paper is available at [3], https://arxiv.org/abs/1801.02358.

Acknowledgements We thank the anonymous referees who helped improve the draft of this paper.

\(^1\) This research was partially funded by the Singapore Ministry of Education and the National Research Foundation, also through the Tier 3 Grant “Random numbers from quantum processes”, MOE2012-T3-1-009.

\(^2\) This research was funded by the National Research Foundation, Prime Minister’s Office, Singapore and the Ministry of Education, Singapore.
1 Introduction

A lattice $\mathcal{L}$ is the set of all integer combinations of linearly independent vectors $b_1, \ldots, b_n \in \mathbb{R}^d$, $\mathcal{L} = \mathcal{L}(b_1, \ldots, b_n) := \{ \sum_{i=1}^{n} z_i b_i : z_i \in \mathbb{Z} \}$.

We call $n$ the rank of the lattice, and $d$ the dimension of the lattice. The matrix $B = (b_1, \ldots, b_n)$ is called a basis of $\mathcal{L}$, and we write $\mathcal{L}(B)$ for the lattice generated by $B$. A lattice is said to be full-rank if $n = d$. In this work, we will only consider full-rank lattices unless otherwise stated.

The two most important computational problems on lattices are the Shortest Vector Problem (SVP) and the Closest Vector Problem (CVP). Given a basis for a lattice $\mathcal{L} \subseteq \mathbb{R}^d$, SVP asks us to compute a non-zero vector in $\mathcal{L}$ of minimal length, and CVP asks us to compute a lattice vector at a minimum distance to a target vector $t$. Typically the length/distance is defined in terms of the $\ell_p$ norm for some $p \in [1, \infty]$, such that $\|x\|_p := (|x_1|^p + |x_2|^p + \cdots + |x_d|^p)^{1/p}$ for $1 \leq p < \infty$, and $\|x\|_\infty := \max_{1 \leq i \leq d} |x_i|$.

The most popular of these, and the most well studied is the Euclidean norm, which corresponds to $p = 2$. Starting with the seminal work of [18], algorithms for solving these problems either exactly or approximately have been studied intensely. Some classic applications of these algorithms are in factoring polynomials over rationals [18], integer programming [19], cryptanalysis [22], checking the solvability by radicals [17], and solving low-density subset-sum problems [12]. More recently, many powerful cryptographic primitives have been constructed whose security is based on the worst-case hardness of these or related lattice problems (see for example [24] and the references therein).

One recent application that is based on the hardness of SVP in the $\ell_\infty$ norm is a recent signature scheme by Ducas et al. [13]. For the security of their signature scheme, the authors choose parameters under the assumption that SVP in the $\ell_\infty$ norm for an appropriate dimension is infeasible. Due to lack of sufficient work on the complexity analysis of SVP in the $\ell_\infty$ norm, they choose parameters based on the best known algorithms for SVP in the $\ell_2$ norm (which are variants of the algorithm from [23]). The rationale for this is that SVP in $\ell_\infty$ norm is likely harder than in the $\ell_2$ norm. Our results in this paper show that this assumption by Ducas et al. [13] is correct, and perhaps too generous. In particular, we show that the space and time complexity of the $\ell_\infty$ version of [23] is at most $(4/3)^n$ and $(4/3)^{2n}$ respectively, which is significantly larger than the best known algorithms for SVP in the $\ell_2$ norm.

The closest vector problem in the $\ell_\infty$ norm is particularly important since it is equivalent to the integer programming problem [14]. The focus of this work is to study the complexity of the closest vector problem and the shortest vector problem in the $\ell_\infty$ norm.

1.1 Prior Work

1.1.1 Algorithms in the Euclidean Norm

The fastest known algorithms for solving these problems run in time $2^{\Omega(n)}$, where $n$ is the rank of the lattice and $c$ is some constant. The first algorithm to solve SVP in time exponential in the dimension of the lattice was given by Ajtai, Kumar, and Sivakumar [5] who devised a method based on “randomized sieving,” whereby exponentially many randomly generated lattice vectors are iteratively combined to create shorter and shorter vectors, eventually resulting in the shortest vector in the lattice. Subsequent work has resulted in improvement of their sieving techniques thereby improving the constant $c$ in the exponent, and the current
fastest provable algorithm for exact SVP runs in time $2^{n+o(n)}$ [1, 4], and the fastest algorithm that gives a constant approximation runs in time $2^{0.802n+o(n)}$ [20]. The fastest heuristic algorithm that is conjectured to solve SVP in practice runs in time $(3/2)^{n/2}$ [9].

The CVP is considered a harder problem than SVP since there is a simple dimension and approximation-factor preserving reduction from SVP to CVP [15]. Based on a technique due to Kannan [16], Ajtai, Kumar, and Sivakumar [6] gave a sieving based algorithm that gives a $1 + \alpha$ approximation of CVP in time $(2 + 1/\alpha)^{O(n)}$. Later exact exponential time algorithms for CVP were discovered [21, 2]. The current fastest algorithm for CVP runs in time $2^{n+o(n)}$ and is due to [2].

1.1.2 Algorithms in Other $\ell_p$ Norms

Blomer and Naewe [11], and then Arvind and Joglekar [7] generalised the AKS algorithm [5] to give exact algorithms for SVP that run in time $2^{O(n)}$. Additionally, [11] gave a $1 + \varepsilon$ approximation algorithm for CVP for all $\ell_p$ norms that runs in time $(2 + 1/\varepsilon)^{O(n)}$. For the special case when $p = \infty$, Eisenbrand et al. [14] gave a $2^{O(n)} \cdot (\log(1/\varepsilon))^n$ algorithm for $(1 + \varepsilon)$-approx CVP.

1.2 Our contribution

1.2.1 Provable Algorithms

We modify the sieving algorithm by [5, 6] for SVP and approximate CVP for the $\ell_\infty$ norm that results in substantial improvement over prior results. Before describing our idea, we give an informal description of the sieving procedure of [5, 6]. The algorithm starts by randomly generating a set $S$ of $N \in 2^{O(n)}$ lattice vectors of length at most $R \in 2^{O(n)}$. It then runs a sieving procedure a polynomial number of times. In the $i$th iteration the algorithm starts with a list $S$ of lattice vectors of length at most $R_{i-1} \approx \gamma^{i-1}R$, for some parameter $\gamma \in (0, 1)$. The algorithm maintains and updates a list of “centres” $C$, which is initialised to be the empty set. Then for each lattice vector $y$ in the list, the algorithm checks whether there is a centre $c$ at distance at most $\gamma \cdot R_{i-1}$ from this vector. If there exists such a centre pair, then the vector $y$ is replaced in the list by $y - c$, and otherwise it is deleted from $S$ and added to $C$. This results in $N_{i-1} - |C|$ lattice vectors which are of length at most $R_i \approx \gamma R_{i-1}$, where $N_{i-1}$ is the number of lattice vectors at the end of $i - 1$ sieving iterations. We mention here that this description hides many details and in particular, in order to show that this algorithm eventually obtains the shortest vector, we need to add a little perturbation to the lattice vectors to start with. The details can be found in Section 3.

A crucial step in this algorithm is to find a vector $c$ from the list of centers that is close to $y$. This problem is called the nearest neighbor search (NNS) problem and has been well studied especially in the context of heuristic algorithms for SVP (see [9] and the references therein). A trivial bound on the running time for this is $|S| \cdot |C|$, but the aforementioned heuristic algorithms have spent considerable effort trying to improve this bound under reasonable heuristic assumptions. Since they require heuristic assumptions, such improved algorithms for the NNS have not been used to improve the provable algorithms for SVP.

We make a simple but powerful observation that for the special case of the $\ell_\infty$ norm, if we partition the ambient space $[-R, R]^n$ into $([-R, -R + \gamma \cdot R), [-R + \gamma \cdot R, -R + 2\gamma \cdot R), \ldots, [-R + \frac{1}{2} \cdot \gamma \cdot R, R)]^n$, then it is easy to see that each such partition will contain at most one centre. Thus, to find a centre at $\ell_\infty$ distance $\gamma \cdot R$ from a given vector $y$, we only need to find the partition in which $y$ belongs, and then check whether this partition contains a centre. This can be easily done by checking the interval in which each co-ordinate
of $y$ belongs. This drastically improves the running time for the sieving procedure in the SVP algorithm from $|S| \cdot |C|$ to $|S| \cdot n$. Notice that we cannot expect to improve the time complexity beyond $O(|S|)$.

This same idea can also be used to obtain significantly faster approximation algorithms for both SVP and CVP. It must be noted here that the prior provable algorithms using AKS sieve lacked an explicit value of the constant in the exponent for both space and time complexity and they used a quadratic sieve. Our modified sieving procedure is linear in the size of the input list and thus yields a faster algorithm compared to the prior algorithms. In order to get the best possible running time, we optimize several steps specialized to the case of $\ell_\infty$ norm in the analysis of the algorithms. See Theorems 15, 17, and 18 for explicit running times and a detailed description.

Just to emphasise that our results are nearly the best possible using these techniques, notice that for a large enough constant $\tau$, we obtain a running time (and space) close to $3^n$ for $\tau$-approximate SVP. To put things in context, the best algorithm [28] for a constant approximate SVP in the $\ell_2$ norm runs in time $2^{0.802n}$ and space $2^{0.401n}$. Their algorithm crucially uses the fact that $2^{0.401n}$ is the best known upper bound for the kissing number of the lattice (which is the number of shortest vectors in the lattice) in $\ell_2$ norm. However, for the $\ell_\infty$ norm, the kissing number is $3^n$ for $\mathbb{Z}^n$. So, if we would analyze the algorithm from [28] for the $\ell_\infty$ norm (without our improvement), we would obtain a space complexity $3^n$, but time complexity $9^n$.

1.2.2 Heuristic Algorithms

In each sieving step of the algorithm from [5], the length of the lattice vectors reduce by a constant factor. It seems like if we continue to reduce the length of the lattice vectors until we get vectors of length $\lambda_1$ (where $\lambda_1$ is the length of the shortest vector), we should obtain the shortest vector during the sieving procedure. However, there is a risk that all vectors output by this sieving procedure are copies of the zero vector and this is the reason that the AKS algorithm [5] needs to start with much more vectors in order to provably argue that we obtain the shortest vector.

Nguyen and Vidick [23] observed that this view is perhaps too pessimistic in practice, and that the randomness in the initial set of vectors should ensure that the basic sieving procedure should output the shortest vector for most lattices, and in particular if the lattice is chosen randomly as is the case in cryptographic applications. The main ingredient to analyze the space and time complexity of their algorithm is to compute the expected number of centres necessary so that any point in $S$ of length at most $R_i - 1$ is at a distance of at most $\gamma \cdot R_i - 1$ from one of the centres. This number is roughly the reciprocal of the fraction of the ball $B$ of radius $R_i - 1$ centred at the origin covered by a ball of radius $\gamma \cdot R_i - 1$ centred at a uniformly random point in $B$. Here $R_i - 1$ is the maximum length of a lattice vector in $S$ after $i - 1$ sieving iterations.

In this work, we show that the heuristic algorithm of [23] can also be analyzed for the $\ell_\infty$ norm under similar assumptions. The main technical contribution in order to analyze the time and space complexity of this algorithm is to compute the expected fraction of an $\ell_\infty$ ball $B(\infty)$ of radius $R_i - 1$ centered at the origin covered by an $\ell_\infty$ ball of radius $\gamma \cdot R_i - 1$ centered at a uniformly random point in $B(\infty)$.

In order to improve the running time of the NV sieve [23], a modified two-level sieve was introduced by Wang et al. [27]. Here they first partition the lattice into sets of vectors of larger norm and then within each set they carry out a sieving procedure similar to [23]. We have analyzed this in the $\ell_\infty$ norm and obtain algorithms much faster than the provable
algorithms. In particular, our two-level sieve algorithm runs in time $2^{0.62n}$. We would like to mention here that our result does not contradict the near $2^n$ lower bound for SVP obtained by [10] under the strong exponential time hypothesis. The reason for this is that the lattice obtained in the reduction in [10] is not a full-rank lattice, and has a dimension significantly larger than the rank $n$ of the lattice. Moreover, as mentioned earlier, the heuristic algorithm is expected to work for a random looking lattice but might not work for all lattices.

Due to space constraints, we have deferred some descriptions and analysis to the full version of this paper [3].

1.3 Open problems

It would be interesting to see if such partitioning technique can be done for other norms or combined with heuristic algorithms like NNS, to yield better performance for sieving algorithms. We do not know if other provable algorithms like those based on Discrete Gaussian sampling [1, 2], Voronoi cells [21] or other heuristic algorithms can be analysed in other non-Euclidean norms. Another direction would be to understand the change in time and space complexity as the number of levels for multi-level sieve increases.

1.4 Organization of the paper

In Section 2 we give some basic definitions and results used in this paper. In Section 3 we introduce our sieving procedure and apply it to provably solve exact SVP$((\infty))$. In Section 4 we describe approximate algorithms for SVP$((\infty))$ and CVP$((\infty))$ using our sieving technique. In Section 5 we talk about heuristic sieving algorithms for SVP$((\infty))$.

2 Preliminaries

2.1 Notations

We write $\ln$ for natural logarithm and $\log$ for logarithm to the base 2.

► **Fact 1.** For $x \in \mathbb{R}^n$ $\|x\|_p \leq \|x\|_2$ and $\frac{1}{\sqrt{p}} \|x\|_p \leq \|x\|_2 \leq \|x\|_p$ for $1 \leq p < 2$.

► **Definition 2 (Ball).** A (closed) ball of radius $r$ and centre at $x \in \mathbb{R}^n$, is the set of all points whose distance (in $\ell_p$ norm) from $x$ is at most $r$. $B^{(p)}(x, r) = \{y \in \mathbb{R}^n : \|y - x\|_p \leq r\}$.

The following result gives a bound on the size of intersection of two balls of a given radius in the $\ell_\infty$ norm.

► **Lemma 3.** Let $v = (v_1, v_2, \ldots, v_n) \in \mathbb{R}^n$, and let $a > 0$ be such that $2a \geq \|v\|_\infty$. Let $D = B^{(\infty)}(0, a) \cap B^{(\infty)}(v, a)$. Then, $|D| = \prod_{i=1}^n (2a - |v_i|)$.

Proof. It is easy to see that the intersection of two balls in the $\ell_\infty$ norm, i.e., hyperrectangles, is also a hyperrectangle. For all $i$, the length of the $i$-th side of this hyperrectangle is $2a - |v_i|$. The result follows.

2.2 Lattice

► **Definition 4.** A lattice $L$ is a discrete additive subgroup of $\mathbb{R}^n$. Each lattice has a basis $B = [b_1, b_2, \ldots b_n]$, where $b_i \in \mathbb{R}^n$ and $L = L(B) = \left\{ \sum_{i=1}^n x_i b_i : x_i \in \mathbb{Z}, \quad i = 1, \ldots, n \right\}$.
For algorithmic purposes we can assume that $L \subseteq \mathbb{Q}^d$.

Definition 5. For any lattice basis $B$ we define the fundamental parallelepiped as:
$$\mathcal{P}(B) = \{Bx : x \in [0, 1)^n\}$$

If $y \in \mathcal{P}(B)$ then $\|y\|_p \leq n\|B\|_p$ as can be easily seen by triangle inequality. For any $z \in \mathbb{R}^n$ there exists a unique $y \in \mathcal{P}(B)$ such that $z - y \in L(B)$. This vector is denoted by $y \equiv z \mod B$ and it can be computed in polynomial time given $B$ and $z$.

Definition 6. For $i \in [n]$, the first successive minimum is defined as the length of the shortest non-zero vector in the lattice:
$$\lambda_1^{(i)}(L) = \min\{\|v\|_p : v \in L \setminus \{0\}\}$$

We consider the following lattice problems. In all the problems defined below $c \geq 1$ is some arbitrary approximation factor. We drop the subscript for exact versions (i.e. $c = 1$).

1. Shortest Vector Problem ($\text{SVP}_c^{(p)}$) Given a lattice $L$, find a vector $v \in L \setminus \{0\}$ such that $\|v\|_p \leq c\|u\|_p$ for any other $u \in L \setminus \{0\}$.

2. Closest Vector Problem ($\text{CVP}_c^{(p)}$) Given a lattice $L$ with rank $n$ and a target vector $t \in \mathbb{R}^n$, find $v \in L$ such that $\|v - t\|_p \leq c\|w - t\|_p$ for all other $w \in L$.

Lemma 7. The LLL algorithm [18] can be used to solve $\text{SVP}_{2^{\gamma n}}^{(p)}$ in polynomial time.

The following result shows that in order to solve $\text{SVP}_{1+\epsilon}^{(p)}$, it is sufficient to consider the case when $2 \leq \lambda_1^{(p)}(L) < 3$. This is done by appropriately scaling the lattice.

Lemma 8 (Lemma 4.1 in [11]). For all $\ell_p$ norms, if there is an algorithm $A$ that for all lattices $L$ with $2 \leq \lambda_1^{(p)}(L) < 3$ solves $\text{SVP}_{1+\epsilon}^{(p)}$ in time $T = T(n, b, \epsilon)$, then there is an algorithm $A'$ that solves $\text{SVP}_{1+\epsilon}^{(p)}$ for all lattices in time $O(nT + n^2b)$.

Thus henceforth we assume $2 \leq \lambda_1^{(\infty)}(L) < 3$.

3 A faster algorithm for $\text{SVP}^{(\infty)}$

In this section we present an algorithm for $\text{SVP}^{(\infty)}$ that uses the framework of AKS algorithm [5] but uses a different sieving procedure that yields a faster running time. Using Lemma 7, we can obtain an estimate $\lambda^* \leq \lambda_1^{(\infty)}(L)$ such that $\lambda_1^{(\infty)}(L) \leq \lambda^* \leq 2^n \cdot \lambda_1^{(\infty)}(L)$. Thus, if we try different values of $\lambda = (1 + 1/n)^{-1} \lambda^*$, for $0 \leq i \leq 10n^2$, then for one of them, we have $\lambda_1^{(\infty)}(L) \leq \lambda \leq (1 + 1/n) \cdot \lambda_1^{(\infty)}(L)$. For this reason, we assume that we know a lower bound $\lambda$ of the length of the shortest vector in $L$, which is correct up to a factor $1 + 1/n$.

As in the AKS algorithm, we start by generating a set $S$ of many vector pairs $(e, y)$, where the perturbation vectors $e$ are uniformly sampled from $B_1^{(\infty)}(\xi\lambda)$ ($\xi > 1/2$), and $y \in e \mod \mathcal{P}(B)$ which has length at most $R$, where $R \leq n \max, \|b_i\|$ and $y - e \in L$. The desired situation is that after a polynomial number of sieving iterations (sieve) we are left with a set of vector pairs $(e', y')$ such that $y' - e' \in L \cap B_1^{(\infty)}(O(\lambda_1^{(\infty)}(L)))$. Finally we take pair-wise differences of the lattice vectors corresponding to the remaining vector pairs and output the one with the smallest non-zero norm. It was shown in [5] that with overwhelming probability, this algorithm outputs the shortest vector in the lattice.

An iteration of the sieving procedure on the pairs of vectors $S$ does the following. We partition the interval $[-R, R]$ into $\ell = 1 + \left\lfloor \frac{2}{\gamma} \right\rfloor$ intervals of length $\gamma R$. The intervals are $[-R, -R + \gamma R], [-R + \gamma R, -R + 2\gamma R], \ldots, [-R + (\ell - 1)\gamma R, R]$. (Note that the last interval may be smaller than the rest.) The ball $[-R, R]^n$ can thus be partitioned into $\left(1 + \left\lfloor \frac{1}{\gamma} \right\rfloor\right)^n$
Algorithm 1: An exact algorithm for $\text{SVP}^{(\infty)}$.

**Input:** (i) A basis $B = [b_1, \ldots, b_n]$ of a lattice $\mathcal{L}$, (ii) $0 < \gamma < 1$, (iii) $\xi > 1/2$, (iv) $\lambda \approx \lambda_1^{(\infty)}(\mathcal{L})$, (v) $N \in \mathbb{N}$

**Output:** A shortest vector of $\mathcal{L}$

1. $S \leftarrow \emptyset$ ;
2. for $i = 1$ to $N$ do
   3. $e_i \leftarrow \text{uniform } B_1^{(\infty)}(0, \xi \lambda) ;$
   4. $y_i \leftarrow e_i \mod \mathcal{P}(B) ;$
   5. $S \leftarrow S \cup \{(e_i, y_i)\} ;$
3. end
4. $R \leftarrow n \max_i \|b_i\|_\infty ;$
5. for $j = 1$ to $k = \left\lfloor \log_\gamma \left(\frac{\xi}{R(1-\gamma)}\right) \right\rfloor$ do
   6. $S \leftarrow \text{sieve}(S, \gamma, R, \xi) ;$
   7. $R \leftarrow \gamma R + \xi \lambda ;$
7. end
8. Compute the non-zero vector $v_0$ in $\{(y_i - e_i) - (y_j - e_j) : (e_i, y_i), (e_j, y_j) \in S\}$ with the smallest $\ell_\infty$ norm ;
9. return $v_0 ;$

regions, such that no two vectors in a region are at a distance greater than $\gamma R$ in the $\ell_\infty$ norm. The sieving procedure maintains an $n$-dimensional array with an entry corresponding to each of these $\left(1 + \frac{2}{\gamma}\right)^n$ regions. Each position in the array contains the description of one pair $(e, y) \in S$ called a center, if $y$ belongs to that region. For every other vector pair $(e, y) \in S$ that is not a center, we find the corresponding region and hence the corresponding center $(e_c, c)$ such that $\|y - c\|_\infty \leq \gamma R$. We then add $(e, y + c - e_c)$ to the output list $S'$. Finally we return $S'$. It is easy to see that the number of center pairs in each iteration is at most $|C| \leq 2^{n} \gamma$ where $c = \log \left(1 + \frac{2}{\gamma}\right)$.

**Claim 9.** The following two invariants are maintained in Algorithm 1:
1. $\forall (e, y) \in S, \quad y - e \in \mathcal{L}$
2. $\forall (e, y) \in S, \quad \|y\|_\infty \leq R$

Since the length of the vectors decrease until $R > \gamma R + \xi \lambda$, the following is easy to see.

**Lemma 10.** At the end of $k$ iterations in Algorithm 1 the length of lattice vectors $\|y - e\|_\infty \leq \left(\frac{2}{1-\gamma}\right)^k + \frac{\gamma}{\gamma n(1-\gamma)} := R'$.

Assuming $\lambda_1^{(\infty)} \leq \lambda \leq \lambda_1^{(\infty)}(1 + 1/n)$ we get an upper bound on the number of lattice vectors of length at most $R'$, i.e. $|B_n^{(\infty)}(R') \cap \mathcal{L}| \leq 2^{\frac{\gamma n + o(n)}{1-\gamma}}$, where $c_0 = \log \left(1 + \frac{2(\frac{2}{1-\gamma})}{\frac{2}{1-\gamma}}\right)$.

The above lemma along with the invariants imply that at the beginning of step 12 in Algorithm 1 we have “short” lattice vectors with norm bounded by $R'$. Using the randomness in the sampling of the initial set of vectors, we want to ensure that we do not end up with all zero vectors at the end of the sieving iterations. For this we use the idea of perturbing the vectors due to Ajtai, Kumar, Sivakumar, and the current formulation by Regev [26].

Let $u \in \mathcal{L}$ such that $\|u\|_\infty = \lambda_1^{(\infty)}(\mathcal{L})$ (where $2 < \lambda_1^{(\infty)}(\mathcal{L}) \leq 3$), $D_1 = B_n^{(\infty)}(\xi \lambda) \cap B_{n}^{(\infty)}(-u, \lambda)$ and $D_2 = B_{n}^{(\infty)}(\xi \lambda) \cap B_{n}^{(\infty)}(u, \lambda)$. Define a bijection $\sigma$ on $B_n^{(\infty)}(\xi \lambda)$ that maps $D_1$ to $D_2$, $D_2 \setminus D_1$ to $D_1 \setminus D_2$ and $B_{n}^{(\infty)}(\xi \lambda) \setminus (D_1 \cup D_2)$ to itself.
Improved Algorithms for SVP and CVP in the Infinity Norm

For the analysis of the algorithm, we assume that for each perturbation vector $e$ chosen by our algorithm, we replace $e$ by $\sigma(e)$ with probability $1/2$ and it remains unchanged with probability $1/2$. We call this procedure tossing the vector $e$. Further, we assume that this replacement of the perturbation vectors happens at the step where for the first time this has any effect on the algorithm. In particular, in the sieving algorithm, after we have identified a centre $(e_c, c)$ we apply $\sigma$ on $e_c$ with probability $1/2$. Then at the beginning of step 12 in Algorithm 1 we apply $\sigma$ to $e$ for all pairs $(e, y) \in S$. The distribution of $y$ remains unchanged by this procedure because $y \equiv e \mod \mathcal{P}(B)$ and $y - e \in \mathcal{L}$. A somewhat more detailed explanation of this can be found in the following result of [11].

Lemma 11 (Theorem 4.5 in [11] (re-stated)). The modification outlined above does not change the output distribution of the actual procedure.

The following lemma will help us estimate the number of vector pairs to sample at the beginning of the algorithm.

Lemma 12 (Lemma 4.7 in [11]). Let $N \in \mathbb{N}$ and $q$ denote the probability that a random point in $B_n^{(\infty)}(\xi \lambda)$ is contained in $D_1 \cup D_2$. If $N$ points $x_1, \ldots, x_N$ are chosen uniformly at random in $B_n^{(\infty)}(\xi \lambda)$, then with probability larger than $1 - \frac{1}{qN}$, there are at least $qN$ points $x_i \in \{x_1, \ldots, x_N\}$ with the property $x_i \in D_1 \cup D_2$.

Using Lemma 3, it can be shown that $q \geq 2^{-c_s n}$ where $c_s = -\log \left( 1 - \frac{1}{r} \right)$. Thus with probability at least $1 - \frac{4}{qN}$, we have at least $2^{-c_s n} N$ pairs $(e_i, y_i)$ before the sieving iterations such that $e_i \in D_1 \cup D_2$.

Lemma 13. If $N \geq \frac{2}{q} (k|C| + 2^{c_s n} + 1)$, then with probability at least $1/2$ Algorithm 1 outputs a shortest non-zero vector in $\mathcal{L}$ with respect to $\ell_\infty$ norm.

Proof. Of the $N$ vector pairs $(e, y)$ sampled in steps 2-6 of Algorithm 1, we consider those such that $e \in (D_1 \cup D_2)$. We have already seen there are at least $\frac{4N}{q}$ such pairs with probability at least $1 - \frac{4}{q}$. We remove $|C|$ vector pairs in each of the $k$ sieve iterations. So at step 12 of Algorithm 1 we have $N' \geq 2^{c_s n} + 1$ pairs $(e_i, y_i)$ to process.

By Lemma 10 each of them is contained within a ball of radius $R'$ which can have at most $2^{c_s n}$ lattice vectors. So there exists at least one lattice vector $w$ for which the perturbation is in $D_1 \cup D_2$ and it appears twice in $S$ at the beginning of step 12. With probability $1/2$ it remains $w$ or with the same probability it becomes either $w + u$ or $w - u$. Thus after taking pair-wise difference at step 12 with probability at least $1/2$ we find the shortest vector. ▶

Thus, the space complexity of our algorithm is $N \cdot \text{poly}(n)$, and the time complexity for the sieving step is $N \cdot \text{poly}(n)$ and for computing the pairwise differences at the end is $2^{2c_s n} \cdot \text{poly}(n)$, thus giving the following result.

Theorem 14. Let $\gamma \in (0, 1)$, and let $\xi > 1/2$. Given a full rank lattice $\mathcal{L} \subset \mathbb{Q}^n$ there is a randomized algorithm for SVP$^{(\infty)}$ with success probability at least $1/2$, space complexity at most $2^{c_{\text{space}} n + o(n)}$ and running time at most $2^{c_{\text{time}} n + o(n)}$, where $c_{\text{space}} = c_s + \max(c_c, c_b)$ and $c_{\text{time}} = \max(c_{\text{space}}, 2c_b)$, where $c_s = -\log \left( 1 + \left[ \frac{1}{\gamma} \right] \right)$, $c_c = -\log \left( 1 - \frac{1}{2^r} \right)$ and $c_b = \log \left( 1 + \left[ \frac{2^{r(2-\gamma)}}{1-\gamma} \right] \right)$. ▶
3.1 Improvement using the birthday paradox

The crucial step that ensures that Algorithm 1 outputs a shortest vector in the lattice is that at step 12, we should have enough vectors to make sure that two vectors are equal (before the tossing step). Pujol and Stehle [25] observed that by the birthday paradox, we only need $2^{2\xi} \in 2 + o(n)$ independent and identically distributed vectors to ensure this. Though their idea was described for the $\ell_2$ norm, we show that the idea can be used to improve the time and space complexity of our algorithm for the $\ell_\infty$ norm [3]. We thus obtain the following result.

**Theorem 15.** Let $\gamma \in (0, 1)$, and let $\xi > 1/2$. Given a full rank lattice $L \subset \mathbb{Q}^n$ there is a randomized algorithm for $\text{SVP}^{(\infty)}$ with success probability at least 1/2, space complexity at most $2^{c_{\text{space}} n + o(n)}$ and running time at most $2^{c_{\text{time}} n + o(n)}$, where $c_{\text{space}} = c_s + \max(c_c, \frac{\gamma}{2})$ and $c_{\text{time}} = \max(c_{\text{space}}, c_b)$, where $c_c = \log \left(1 + \left\lfloor \frac{1}{\gamma} \right\rfloor\right)$, $c_s = -\log \left(1 - \frac{1}{\gamma}\right)$ and $c_b = \log \left(1 + \frac{2^{2(2-\gamma)} - 1}{\gamma}ight)$.

In particular for $\gamma = 0.67$ and $\xi = 0.868$ the algorithm has time and space complexity $2^{2.82n + o(n)}$.

4 Faster Approximation Algorithms

4.1 Algorithm for Approximate SVP

Notice that Algorithm 1, at the end of the sieving procedure, obtains lattice vectors of length at most $R' = \frac{\xi(2-\gamma)\lambda}{1-\gamma} + O(\lambda/n)$. So, as long as we can ensure that one of the vectors obtained at the end of the sieving procedure is non-zero, we obtain a $\tau = \frac{\xi(2-\gamma)\lambda}{1-\gamma} + o(1)$-approximation of the shortest vector. Consider a new algorithm $A$ that is identical to Algorithm 1, except that Step 12 is replaced by the following:

- Find a non-zero vector $v_0$ in $\{(y_i - e_i) : (e_i, y_i) \in S\}$.

We now show that if we start with sufficiently many vectors, we must obtain a non-zero vector.

**Lemma 16.** If $N \geq \frac{2}{\gamma}(k|C| + 1)$, then with probability at least 1/2 Algorithm $A$ outputs a non-zero vector in $L$ of length at most $\frac{\xi(2-\gamma)\lambda}{1-\gamma} + O(\lambda/n)$ with respect to $\ell_\infty$ norm.

**Proof.** Of the $N$ vector pairs $(e, y)$ sampled in steps 2-6 of Algorithm $A$, we consider those such that $e \in (D_1 \cup D_2)$. We have already seen there are at least $\frac{N}{2}$ such pairs with probability at least $1 - \frac{1}{2N}$. We remove $|C|$ vector pairs in each of the $k$ sieve iterations. So at step 12 of Algorithm 1 we have $N' \geq 1$ pairs $(e, y)$ to process.

With probability 1/2, $e$, and hence $w = y - e$ is replaced by either $w + u$ or $w - u$. Thus, the probability that this vector is the zero vector is at most 1/2.

We thus obtain the following:

**Theorem 17.** Let $\gamma \in (0, 1)$ and $\xi > 1/2$. Given a full rank lattice $L \subset \mathbb{Q}^n$ there is a randomized algorithm that, for $\tau = \frac{\xi(2-\gamma)\lambda}{1-\gamma} + o(1)$, approximates $\text{SVP}^{(\infty)}$ with success probability at least 1/2, space and time complexity $2^{c_s + c_b} n + o(n)$, where $c_c = \log \left(1 + \left\lfloor \frac{1}{2\gamma} \right\rfloor\right)$, and $c_s = -\log \left(1 - \frac{1}{2\gamma}\right)$.

In particular, for $\gamma = 2/3 + o(1)$, $\xi = \tau/4$, the algorithm runs in time $3^n \cdot \left(\frac{\tau}{\tau - \gamma}\right)^n$.  

ISAAC 2018
4.2 Algorithm for Approximate CVP

Given a lattice \( \mathcal{L} \) and a target vector \( t \), let \( d \) denote the distance of the closest vector in \( \mathcal{L} \) to \( t \). Just as in Section 3, we assume that we know the value of \( d \) within a factor of \( 1 + 1/\pi \). We can get rid of this assumption by using Babai’s [8] algorithm to guess the value of \( d \) within a factor of \( 2^n \), and then run our algorithm for polynomially many values of \( d \) each within a factor \( 1 + 1/\pi \) of the previous one.

For \( \tau > 0 \) define the following \( (n + 1) \)-dimensional lattice: \( \mathcal{L}' = \mathcal{L}(\{(v, 0) : v \in \mathcal{L} \} \cup \{(t, \tau d/2)\}) \). Let \( z^* \in \mathcal{L} \) be the lattice vector closest to \( t \). Then \( u = (z^* - t, -\tau d/2) \in \mathcal{L}' \setminus \mathcal{L} \) for some \( k \in \mathbb{Z} \).

We sample \( N \) vector pairs \( (e, y) \in \mathcal{B}_n^\infty(\xi d) \times \mathcal{P}(\mathcal{B}') \), like in steps 2-6 of Algorithm 1, where \( \mathcal{B}' = [(b_1, 0), \ldots, (b_n, 0), (t, \tau d/2)] \) is a basis for \( \mathcal{L}' \). Next we run a polynomial number of iterations of the sieving algorithm (sieve) to get a number of vector pairs such that \( |v|_\infty \leq R = \frac{2d}{1 - \gamma} + o(1) \).

From Lemma 10 we have seen that after \( \lceil \log_2 \left( \frac{\xi}{nR_0(1-\gamma)} \right) \rceil \) iterations (where \( R_0 = n \cdot \max(|b_i|_\infty) \sqrt{d} \gamma + \frac{4d}{1 - \gamma} \left[ 1 - \frac{\xi}{nR_0(1-\gamma)} \right] \). Thus after the sieving iterations the set \( S' \) consists of vector pairs such that the corresponding lattice vector \( v \) has \( |v|_\infty \leq \frac{2d}{1 - \gamma} + 4d + c = \left( \frac{2 - \gamma}{1 - \gamma} \right) d + o(1) \).

In order to ensure that our sieving algorithm doesn’t return vectors from \((\mathcal{L}, 0) - (kt, k\tau d/2)\) for some \( k \geq 2 \), we choose our parameter as: \( \xi < \frac{(1 - \gamma) d}{2 - \gamma} - o(1) \).

Then every vector has \( |v|_\infty < \tau d \) and so either \( v = \pm(z' - t, 0) \) or \( v = \pm(z - t, -\tau d/2) \) for some lattice vector \( z, z' \in \mathcal{L} \). We denote this set of vectors by \( S'' \).

We need to argue that we must have at least some vectors in \( S'' \setminus (\mathcal{L} \pm t, 0) \) after the sieving iterations. To do so, we again use the tossing argument from Section 3. Let \( z^* \in \mathcal{L} \) be the lattice vector closest to \( t \). Then let \( u = (z^* - t, -\tau d/2) \in S'' \setminus (\mathcal{L} \pm t, 0) \). Let \( D_1 = \mathcal{B}_n^\infty(\xi d) \cap \mathcal{B}_n^\infty(-u, \xi d) \) and \( D_2 = \mathcal{B}_n^\infty(\xi d) \cap \mathcal{B}_n^\infty(u, \xi d) \).

From Lemma 3, we have that the probability \( q \) that a random perturbation vector is in \( D_1 \cup D_2 \) is at least \( 2^{-c_2 n} \left( 1 - \frac{1}{2^\gamma} \right) \), where \( c_2 = -\log \left( 1 - \frac{1}{2^\gamma} \right) \).

Thus, as long as \( \xi > \max(1/2, \tau/4) \), we have at least \( 2^{-c_2 n} n \) pairs \( (e_i, y_i) \) before the sieving iterations such that \( e_i \in D_1 \cup D_2 \).

Thus, using the same argument as in Section 4.1, we obtain the following:

**Theorem 18.** Let \( \gamma \in (0, 1) \), and for any \( \tau > 1 \) let \( \xi > \max(1/2, \tau/4) \). Given a full rank lattice \( \mathcal{L} \subset \mathbb{Q}^n \) there is a randomized algorithm that, for \( \tau = \frac{1}{2 - \gamma} + o(1) \), approximates \( CVP(\infty) \) with success probability at least 1/2, space and time complexity \( 2^{(c_2 + c_3) n + o(n)} \), where \( c_2 = \log \left( 1 + \frac{1}{2^\gamma} \right) \) and \( c_3 = -\log \left( 1 - \frac{1}{2^\gamma} \right) \). In particular, for \( \gamma = 1/2 + o(1) \) and \( \xi = \tau/3 \), the algorithm runs in time \( 4^n \left( \frac{2^\tau}{2^\gamma - 3} \right)^n \).

5 Heuristic algorithm for SVP(\( \infty \))

Nguyen and Vidick [23] introduced a heuristic variant of the AKS sieving algorithm. We have used it to solve \( SVP(\infty) \). A brief outline of the algorithm is given in this section while a more detailed description along with the analysis is deferred to the full version [3].

The basic framework is similar to AKS, except that here we do not work with perturbation vectors. We start with a set \( S \) of uniformly sampled lattice vectors of norm \( 2^{0(n)} \lambda(\infty)(\mathcal{L}) \). These are iteratively fed into a sieving procedure which when provided with a list of lattice vectors...
vectors of norm, say \( R \), will return a list of lattice vectors of norm at most \( \gamma R \). In each iteration of the sieve a number of vectors are identified as centres. If a vector is within distance \( \gamma R \) from a centre, we subtract it from the centre and add the resultant to the output list. The iterations continue till the list \( S \) of vectors currently under consideration is empty. After a linear number of iterations we expect to be left with a list of very short vectors and then we output the one with the minimum norm. Here we have to ensure that we do not end up with a list of all zero-vectors much before we get these short vectors.

So we make the following assumption about the distribution of vectors at any stage of the algorithm.

**Heuristic 19.** At any stage of the algorithm the vectors in \( S \cap B_n^{(\infty)}(\gamma R, R) \) are uniformly distributed in \( B_n^{(\infty)}(\gamma R, R) = \{ x \in \mathbb{R}^n : \gamma R < \|x\|_\infty \leq R \} \).

In the literature, such assumption has been made for \( \ell_2 \) norm. We have extended the same assumption to \( \ell_\infty \) norm, because we could not find evidence that it does not hold here.

Now after each sieving iteration we get a zero vector if there is a “collision” of a vector with a centre vector. With the above assumption we can have following estimate about the expected number of collisions.

**Lemma 20** ([23]). Let \( p \) vectors are randomly chosen with replacement from a set of cardinality \( N \). Then the expected number of different vectors picked is \( N - N(1 - \frac{1}{n})^p \). So the expected number of vectors lost through collisions is \( p - N + N(1 - \frac{1}{n})^p \).

This number is negligible for \( p << \sqrt{N} \). Since the expected number of lattice points inside a ball of radius \( R/\lambda_1^{(\infty)} \) is \( O(R^n) \), the effect of collisions remain negligible till \( R/\lambda_1^{(\infty)} < |S|^{2/n} \). It can be shown that it is sufficient to take \( |S| \approx (4/3)^n \), which gives \( R/\lambda_1^{(\infty)} \approx 16/9 \). So collisions are expected to become significant only when we already have a good estimate of \( \lambda_1^{(\infty)} \), and even then collisions will imply we had a good proportion of lattice vectors in the previous iteration and thus with good probability we expect to get the shortest vector or a constant approximation of it.

Choosing \( \gamma = 1 - 1/n \), our algorithm has space complexity \( \left( \frac{4}{3} \right)^{n+o(n)} = 2^{0.415n+o(n)} \) and time complexity \( \left( \frac{4}{3} \right)^{2n+o(n)} = 2^{0.83n+o(n)} \).

In order to improve the running time, which is mostly dictated by the number of centres, Wang et al. [27] introduced a two-level sieving procedure that improves upon the NV sieve for large \( n \). Here in the first level we identify a set of centres \( C_1 \) and to each \( c \in C_1 \) we associate vectors within a distance \( \gamma_1 R \) from it. Now within each such \( \gamma_1 R \) radius “big ball” we have another set of vectors \( C_2^c \), which we call the second-level centre. From each \( c' \in C_2^c \) we subtract those vectors which are in \( B_n^{(\infty)}(c', \gamma_2 R) \) and add the resultant to the output list.

We have analysed this two-level sieve in the \( \ell_\infty \) norm and also found similar improvement in the running time. For suitable choice of parameters we achieve a space and time complexity of at most \( 2^{0.415n+o(n)} \) and \( 2^{0.62n+o(n)} \) respectively.

References


26 Oded Regev. Lecture notes on lattices in computer science, 2009.
An Adaptive Version of Brandes’ Algorithm for Betweenness Centrality

Matthias Bentert  
Algorithmics and Computational Complexity, Faculty IV, TU Berlin, Germany  
matthias.bentert@tu-berlin.de

Alexander Dittmann  
Algorithmics and Computational Complexity, Faculty IV, TU Berlin, Germany  
alexander.dittmann@campus.tu-berlin.de

Leon Kellerhals  
Algorithmics and Computational Complexity, Faculty IV, TU Berlin, Germany  
leon.kellerhals@tu-berlin.de

André Nichterlein  
Algorithmics and Computational Complexity, Faculty IV, TU Berlin, Germany  
andre.nichterlein@tu-berlin.de

Rolf Niedermeier  
Algorithmics and Computational Complexity, Faculty IV, TU Berlin, Germany  
rolf.niedermeier@tu-berlin.de

Abstract

Betweenness centrality – measuring how many shortest paths pass through a vertex – is one of the most important network analysis concepts for assessing the relative importance of a vertex. The well-known algorithm of Brandes [2001] computes, on an \( n \)-vertex and \( m \)-edge graph, the betweenness centrality of all vertices in \( O(nm) \) worst-case time. In follow-up work, significant empirical speedups were achieved by preprocessing degree-one vertices and by graph partitioning based on cut vertices. We further contribute an algorithmic treatment of degree-two vertices, which turns out to be much richer in mathematical structure than the case of degree-one vertices. Based on these three algorithmic ingredients, we provide a strengthened worst-case running time analysis for betweenness centrality algorithms. More specifically, we prove an adaptive running time bound \( O(kn) \), where \( k < m \) is the size of a minimum feedback edge set of the input graph.

2012 ACM Subject Classification  Theory of computation → Graph algorithms analysis

Keywords and phrases  network science, social network analysis, centrality measures, shortest paths, tree-like graphs, efficient pre- and postprocessing, FPT in P

Digital Object Identifier  10.4230/LIPIcs.ISAAC.2018.36


1 Supported by DFG project FPTinP, NI 369/16.
An Adaptive Version of Brandes’ Algorithm for Betweenness Centrality

1 Introduction

One of the most important building blocks in network analysis is to determine a vertex’s relative importance in the network. A key concept herein is betweenness centrality as introduced in 1977 by Freeman [6]; it measures centrality based on shortest paths. Intuitively, for each vertex, betweenness centrality counts the (relative) number of shortest paths that pass through the vertex. A straightforward algorithm for computing the betweenness centrality on undirected (unweighted) $n$-vertex graphs runs in $\Theta(n^3)$ time, and improving this to $O(n^3 - \varepsilon)$ time for any $\varepsilon > 0$ would break the so-called APSP-conjecture [1]. In 2001, Brandes [3] presented the to date theoretically fastest algorithm, improving the running time to $O(nm)$ for graphs with $m$ edges. As many real-world networks are sparse, this is a far-reaching improvement, having a huge impact also in practice. Newman [9] presented a high-level description of an algorithm for a variant of betweenness centrality running in $O(nm)$ time.

Our work is in line with numerous research efforts concerning the development of algorithms for computing betweenness centrality. Formally, we study the following problem:

**Betweenness Centrality**

**Input:** An undirected graph $G$.

**Task:** Compute the betweenness centrality $C_B(v) := \sum_{s,t \in V(G)} \frac{\sigma_{st}(v)}{\sigma_{st}}$ for each vertex $v \in V(G)$.

Herein, $\sigma_{st}$ is the number of shortest paths in $G$ from vertex $s$ to vertex $t$, and $\sigma_{st}(v)$ is the number of shortest paths from $s$ to $t$ that additionally pass through $v$.

Extending previous, more empirically oriented work of Baglioni et al. [2], Puzis et al. [12], and Sariyüce et al. [13] (see Section 2 for a description of their approaches), our main result is the mathematically rigorous analysis of an algorithm for Betweenness Centrality that runs in $O(kn)$ time, where $k$ denotes the feedback edge number of the input graph $G$. The feedback edge number of $G$ is the minimum number of edges to be deleted from $G$ in order to make it a forest. Clearly, $k = 0$ holds on trees, and $k \leq m$ holds in general. Thus our algorithm is adaptive, i.e., it interpolates between linear time for constant $k$ and the running time of the best unparameterized algorithm for $k$ approaching $m$. Obviously, by depth-first search one can compute $k$ in linear time; however, $k \approx m - n$, so we provide no asymptotic improvement over Brandes’ algorithm for most graphs. When the input graph is very tree-like ($m = n + o(n)$), however, our new algorithm improves on Brandes’ algorithm. Real-world networks showing the relation between PhD candidates and their supervisors [4, 8] or the ownership relation between companies [11] typically have a feedback edge number that is smaller than the number of vertices or edges by orders of magnitude [10]. For roughly half of their networks, $m - n$ is smaller than $n$ by at least one order of magnitude.

Our algorithmic contribution is to complement the works of Baglioni et al. [2], Puzis et al. [12], and Sariyüce et al. [13] by, roughly speaking, additionally dealing with degree-two vertices. These vertices are much harder to cope with and to analyze since, other than degree-one vertices, they may lie on shortest paths between two vertices. Recently, Vella et al. [14] used a heuristic approach to process degree-two vertices for improving the performance of their Betweenness Centrality algorithms on several real-world networks.

---

2 To simplify our matters, we set $\sigma_{st}(v) = 0$ if $v = s$ or $v = t$. This is equivalent to Brandes [3] but differs from Newman [9], where $\sigma_{st}(s) = 1$.

3 Notably, Betweenness Centrality computations have also been studied when the input graph is a tree [15], hinting at the practical relevance of this special case.
Our work is purely theoretical in spirit. Our most profound contribution is to analyze the worst-case running time of the proposed betweenness centrality algorithm based on degree-one-vertex processing [2], usage of cut vertices [12, 13], and our degree-two-vertex processing. To the best of our knowledge, this provides the first proven worst-case “improvement” over Brandes’ upper bound in a relevant special case.

Notation. We use mostly standard graph notation. Given a graph $G$, $V(G)$ and $E(G)$ denote the vertex respectively edge set of $G$ with $n = |V(G)|$ and $m = |E(G)|$. We denote the vertices of degree one, two, and at least three by $V^1$, $V^2$, and $V^3$, respectively. A cut vertex or articulation vertex is a vertex whose removal disconnects the graph. A connected component of a graph is biconnected if it does not contain any cut vertices, and hence, no vertices of degree one. A path $P = v_0 \ldots v_q$ is a graph with $V(P) = \{v_0, \ldots, v_q\}$ and $E(P) = \{(v_i, v_{i+1}) | 0 \leq i < q\}$. The length of the path $P$ is $|E(P)|$. Adding the edge $(v_q, v_0)$ to $P$ gives a cycle $C = v_0 \ldots v_q v_0$. The distance $d_G(s, t)$ between vertices $s, t \in V(G)$ is the length of the shortest path between $s$ and $t$ in $G$. The number of shortest $s$–$t$–paths is denoted by $\sigma_{st}$. The number of shortest $s$–$t$–paths containing some vertex $v$ is denoted by $\sigma_{st}(v)$. We set $\sigma_{st}(v) = 0$ if $s = v$ or $t = v$ (or both). Lastly, for $j \leq k$ we set $[j, k] := \{j, j + 1, \ldots, k\}$.

2 Algorithm overview

In this section, we review our algorithmic strategy to compute the betweenness centrality of each vertex. Before doing so, since we build on the works of Brandes [3], Baglioni et al. [2], Puzis et al. [12], and Sariyüce et al. [13], we first give the high-level ideas behind their algorithmic approaches. Then, we describe the ideas behind our extension. We remark that we assume throughout our paper that the input graph is connected. Otherwise, we can process the connected components one after another.

Existing algorithmic approaches. Brandes [3] developed an $O(nm)$-time algorithm which essentially runs modified breadth-first searches (BFS) from each vertex of the graph. In each of these modified BFS, Brandes’ algorithm computes the “effect” that the starting vertex $s$ of the modified BFS has on the betweenness centrality values of all other vertices. More formally, the modified BFS starting at vertex $s$ computes $\sum_{t \in V(G)} \sigma_{st}(v)/\sigma_{st}$ for each vertex $v \in V(G)$.

Reducing the number of performed modified BFS in Brandes’ algorithm is one way to speed up Brandes’ algorithm. To this end, a popular approach is to remove in a preprocessing step all degree-one vertices from the graph [2, 12, 13]. By repeatedly removing degree-one vertices, whole “pending trees” can be deleted. Considering a degree-one vertex $v$, observe that in each shortest path $P$ starting at $v$, the second vertex in $P$ is the single neighbor $u$ of $v$. Hence, after deleting $v$, one needs to store the information that $u$ had a degree-one neighbor. To this end, one uses for each vertex $w$ a counter which we call $\text{Pen}[w]$ that stores the number of vertices in the subtree pending on $w$ that where deleted before. In contrast to e.g. Baglioni et al. [2], we initialize for each vertex $w \in V$ the value $\text{Pen}[w]$ with one instead of zero (so we count $w$ as well). This simplifies most of our formulas. See Figure 1 for an example of the $\text{Pen}[\cdot]$-values of the vertices at different points in time. This yields the following (weighted) problem variant.
Figure 1 An initial graph where the Pen\{\cdot\}-value of each vertex is 1 (top left) and the same graph after deleting one (top right) or both (bottom left) pending trees using Reduction Rule 1. The labels are the respective Pen\{\cdot\}-values. Subfigure (4.) shows the graph of (3.) after applying Lemma 2 to the only remaining cut vertex of the graph.

Weighted Betweenness Centrality

Input: An undirected graph \( G \) and vertex weights \( \text{Pen}: V(G) \rightarrow \mathbb{N} \).

Task: Compute for each vertex \( v \in V(G) \) the weighted betweenness centrality

\[
C_B(v) := \sum_{s,t \in V(G)} \gamma(s,t,v),
\]

where \( \gamma(s,t,v) := \text{Pen}[s] \cdot \text{Pen}[t] \cdot \sigma_{st}(v)/\sigma_{st} \).

The effect of a degree-one vertex to the betweenness centrality value of its neighbor is captured in the next data reduction rule.

▶ Reduction Rule 1 ([12, 13]). Let \( G \) be a graph, let \( s \in V(G) \) be a degree-one vertex, and let \( v \in V(G) \) be the neighbor of \( s \). Then increase \( \text{Pen}[v] \) by \( \text{Pen}[s] \), increase the betweenness centrality of \( v \) by \( \text{Pen}[s] \cdot \sum_{t \in V(G) \setminus \{s,v\}} \text{Pen}[t] \), and remove \( s \) from the graph.

Hence, the influence of a degree-one vertex to the betweenness centrality of its neighbor can be computed in constant time as \( \sum_{w \in V(G)} \text{Pen}[w] \) can be precomputed once in linear time.

A second approach to speed up Brandes’ algorithm is to split the input graph \( G \) into smaller components and process them separately [12, 13]. This approach is a generalization of the ideas behind removing degree-one vertices and works with cut vertices. The basic observation for this approach is as follows: Consider a cut vertex \( v \) such that removing \( v \) breaks the graph into exactly two connected components \( C_1 \) and \( C_2 \) (the idea generalizes to more components). Obviously, every shortest path \( P \) in \( G \) that starts in \( C_1 \) and ends in \( C_2 \) has to pass through \( v \). For the betweenness centrality values of the vertices inside \( C_1 \) (inside \( C_2 \)) it is not important where exactly \( P \) ends (starts). Hence, for computing the betweenness centrality values of the vertices in \( C_1 \), it is sufficient to know which vertices in \( C_1 \) are adjacent to \( v \) and how many vertices are contained in \( C_2 \). Thus, in a preprocessing step one can just add to \( C_1 \) a copy of the cut vertex \( v \) with \( \text{Pen}[v] \) being increased by the sum of \( \text{Pen}[\cdot] \)-values of the vertices in \( C_2 \) (see Figure 1 (bottom)). The same is done for \( C_2 \). Formally, this is done as follows.

▶ Lemma 2 ([12, 13]). Let \( G \) be a connected graph, \( v \) be a cut vertex such that removing \( v \) yields \( \ell \geq 2 \) connected components \( C_1, \ldots, C_\ell \), and let \( \xi := \text{Pen}[v] \). Then remove \( v \), add a new vertex \( v_i \) to each component \( C_i \), make them adjacent to all vertices in the respective
Computing the betweenness centrality of each connected component independently, increasing the number of maximal induced paths is the same as computing the betweenness centrality in \( G \), that is,

\[
C_B^G(u) = \begin{cases} 
C_B^{C_i}(u), & \text{if } u \in V(C_i) \setminus \{v_i\}; \\
\sum_{i=1}^r (C_B^{C_i}(v_i) + (\text{Pen}[v_i] - \xi) \cdot \sum_{s \in V(C_i) \setminus \{v_i\}} \text{Pen}[s]), & \text{if } u = v.
\end{cases}
\]

Applying the above procedure as preprocessing on all cut vertices and degree-one vertices takes linear time \([13]\) leaves us with biconnected components that we can solve independently. Hence, we assume in the rest of the paper that we are given a vertex-weighted biconnected component.

**Our algorithmic approach.** Starting with a vertex-weighted biconnected graph, our algorithm focuses on degree-two vertices. In contrast to degree-one vertices, degree-two vertices can lie on shortest paths between two other vertices. This difference makes degree-two vertices harder to handle: Removing a degree-two vertex \( v \) in a similar way as done with degree-one vertices (see Reduction Rule 1) affects many other shortest paths that neither start nor end in \( v \). Hence, we deal with degree-two vertices in a different manner. Instead of removing vertices one-by-one, we process multiple degree-two vertices at once. To this end, we use the following definition and exploit that adjacent degree-two vertices behave similarly.

**Definition 3.** Let \( G \) be a graph. A path \( P = v_0 \ldots v_\ell \) is a maximal induced path in \( G \) if \( \ell \geq 2 \) and the inner vertices \( v_1, \ldots, v_{\ell-1} \) all have degree two in \( G \), but the endpoints \( v_0 \) and \( v_\ell \) do not, that is, \( \deg_G(v_1) = \ldots = \deg_G(v_{\ell-1}) = 2 \), \( \deg_G(v_0) \neq 2 \), and \( \deg_G(v_\ell) \neq 2 \). Moreover, \( P^\text{max} \) is the set of all maximal induced paths in \( G \).

Note that if our biconnected graph is a cycle, then it does not contain any maximal induced path. Our algorithm (see Algorithm 1 for the pseudocode) deals with this corner case separately by using a linear-time dynamic programming algorithm for vertex-weighted cycles. Note that the vertices in the cycle can have different betweenness centrality values as they may have different \( \text{Pen}[\cdot] \)-values.

**Proposition 4 \((\star)\).** Let \( G = x_0 \ldots x_q x_0 \) be a cycle. Then, the weighted betweenness centrality of the vertices in \( C \) can be computed in \( O(q) \) time.

The remaining part of the algorithm deals with maximal induced paths. Note that if the (biconnected) graph is not a cycle, then all degree-two vertices are contained in maximal induced paths: If the graph is not a cycle and does not contain degree-one vertices, then the endpoints of each chain of degree-two vertices are vertices of degree at least three. If some degree-two vertex \( v \) was not contained in a maximal induced path, then \( v \) would be contained on a cycle with exactly one vertex of degree at least three. This vertex would be a cut vertex and the graph would not be biconnected; a contradiction.

Using standard arguments, we can show that the number of maximal induced paths is upper-bounded by the minimum of the feedback edge number \( k \) of the input graph and the number \( n \) of vertices. Moreover, one can easily compute all maximal induced paths in linear-time (see Line 6 in Algorithm 1).

\(^4\) Proofs of results marked with \((\star)\) are deferred to the full version.
An Adaptive Version of Brandes’ Algorithm for Betweenness Centrality

Algorithm 1: Computation of betweenness centrality in a biconnected graph.

Input: An undirected biconnected graph $G$ with vertex weights $\text{Pen}: V(G) \rightarrow \mathbb{N}$.
Output: The betweenness centrality values of all vertices.

1. $\text{foreach } v \in V(G) \text{ do } \text{BC}[v] \leftarrow 0$  
   // BC will contain the betweenness centrality values
2. $F \leftarrow$ feedback edge set of $G$  
   // computable in $O(n + m)$ time using BFS
3. if $|F| = 1$ then
   4. update BC for the case that $G$ is a cycle // computable in $O(n)$ time, see Proposition 4
5. else
6.   $\mathcal{P}^{\text{max}} \leftarrow$ all maximal induced paths of $G$  
   // takes $O(n + m)$ time, see Lemma 6
7.   $\text{foreach } s \in V^{\geq 3}(G) \text{ do }$  
       // some precomputations taking $O(|F| n)$ time, see Lemma 10
8.       compute $d_G(s, t)$ and $\sigma_d$ for each $t \in V(G) \setminus \{s\}$
9.       $\text{Inc}[s, t] \leftarrow 2 \cdot \text{Pen}[s] \cdot \text{Pen}[t] / \sigma_d$ for each $t \in V^{=2}(G)$
10.      $\text{Inc}[s, t] \leftarrow \text{Pen}[s] \cdot \text{Pen}[t] / \sigma_d$ for each $t \in V^{=3}(G) \setminus \{s\}$
11.     $\text{foreach } x_0 x_1 \ldots x_q = \mathcal{P}^{\text{max}} \in \mathcal{P}^{\text{max}} \text{ do }$  
          // initialize $W^{\text{left}}$ and $W^{\text{right}}$, in $O(n)$ time
12.        $W^{\text{left}}[x_0] \leftarrow \text{Pen}[x_0]; W^{\text{right}}[x_q] \leftarrow \text{Pen}[x_q]$
13.        for $i = 1$ to $q$ do $W^{\text{left}}[x_i] \leftarrow W^{\text{left}}[x_{i-1}] + \text{Pen}[x_i]$
14.        for $i = q - 1$ to $0$ do $W^{\text{right}}[x_i] \leftarrow W^{\text{right}}[x_{i+1}] + \text{Pen}[x_i]$
15.     $\text{foreach } x_0 x_1 \ldots x_q = \mathcal{P}^{\text{max}} \in \mathcal{P}^{\text{max}} \text{ do }$  
          // case $s \in V^{=2}(\mathcal{P}^{\text{max}})$, see Section 3
16.        /* deal with the case $t \in V^{=2}(\mathcal{P}^{\text{max}})$, see Section 3.1 */
17.        $\text{foreach } y_0 y_1 \ldots y_r = \mathcal{P}^{\text{max}} \in \mathcal{P}^{\text{max}} \setminus \{\mathcal{P}^{\text{max}}\} \text{ do }$
18.            /* update BC for the case $v \in V(\mathcal{P}^{\text{max}}) \cup V(\mathcal{P}^{\text{max}})$ */
19.            $\text{foreach } v \in V(\mathcal{P}^{\text{max}}) \cup V(\mathcal{P}^{\text{max}}) \text{ do }$  
                   $\text{BC}[v] \leftarrow B(t, s, v)$
20.            /* now deal with the case $v \not\in V(\mathcal{P}^{\text{max}}) \cup V(\mathcal{P}^{\text{max}})$ */
21.            update $\text{Inc}[x_0, y_0], \text{Inc}[x_r, y_r], \text{Inc}[x_0, y_r], \text{Inc}[x_0, y_r]$,
22.               and $\text{Inc}[x_0, y_r]$  
23.        /* deal with the case that $t \in V^{=2}(\mathcal{P}^{\text{max}})$, see Section 3.1 */
24.        $\text{foreach } v \in V(\mathcal{P}^{\text{max}}) \text{ do }$  
                   $\text{BC}[v] \leftarrow B(t, s, v) + \gamma(s, t, v)$
25.        update $\text{Inc}[x_0, x_q]$  
                  /* this deals with the case $v \not\in V(\mathcal{P}^{\text{max}})$ */
26.     // perform modified BFS from $s$, see Section 3.2
27.     $\text{foreach } t, v \in V(G) \text{ do }$  
                   $\text{BC}[v] \leftarrow B(t, s, v) + \text{Inc}[s, t] \cdot \sigma_d(v)$
28. return BC.

Lemma 5 (*). Let $G$ be a graph with feedback edge number $k$ containing no degree-one vertices. Then the cardinalities $|V^{=3}(G)|$ and $|\mathcal{P}^{\text{max}}|$ are upper-bounded by $O(\min\{n, k\})$.

Lemma 6 (*). The set $\mathcal{P}^{\text{max}}$ of all maximal induced paths of a graph with $n$ vertices and $m$ edges can be computed in $O(n + m)$ time.

Our algorithm processes the maximal induced paths one by one (see Lines 7 to 22). This part of the algorithm requires its own pre- and postprocessing (see Lines 7 to 14 and Lines 21 to 22 respectively). In the preprocessing, we initialize tables used frequently in the main part (of Section 3). The postprocessing computes the final betweenness centrality values of each vertex as this computation is too time-consuming to be executed for each maximal induced path. When explaining our basic ideas, we will first present the postprocessing as this explains why certain values will be computed during the algorithm.

Recall that we want to compute $\sum_{t \in V(G)} \gamma(s, t, v)$ for each $v \in V(G)$ (see Equation (1)). Using the following observations, we split Equation (1) into different parts:

Observation 7. For $s, t, v \in V(G)$ it holds that $\gamma(s, t, v) = \gamma(t, s, v)$. 
Observation 8 (∗). Let $G$ be a biconnected graph with at least one vertex of degree three. Let $v \in V(G)$. Then,

\[
\sum_{s,t \in V(G)} \gamma(s, t, v) = \sum_{s \in V^{>3}(G), t \in V(G)} \gamma(s, t, v) + \sum_{s \in V^{=2}(G), t \in V^{>3}(G)} \gamma(t, s, v)
\]

\[
+ \sum_{s \in V^{=2}(p_{\text{max}}^{1}), t \in V^{=2}(p_{\text{max}}^{2})} \sum_{p_{\text{max}} \in p_{\text{max}}^{\ast}} \gamma(s, t, v) + \sum_{s, t \in V^{=2}(p_{\text{max}}^{\ast})} \gamma(s, t, v).
\]

In the remaining graph, by Lemma 5, there are $O(\min\{k, n\})$ vertices of degree at least three and $O(k)$ maximal induced paths. This implies that we can afford to run the modified BFS (similar to Brandes’ algorithm) from each vertex $s \in V^{\geq 3}(G)$ in $O(\min\{k, n\} \cdot (n + k)) = O(kn)$ time. This computes the first summand and, by Observation 7, also the second summand in Observation 8. However, we cannot afford to run such a BFS from every vertex of degree two. Thus we need to compute the third and fourth summand differently.

To this end, note that $\sigma_{st}(v)$ is the only term in $\gamma(s, t, v)$ that depends on $v$. Our goal is then to precompute $\gamma(s, t, v)/\sigma_{st}(v) = \text{Pen}[s] \cdot \text{Pen}[t]/\sigma_{st}$ for as many vertices as possible. Hence, we store precomputed values in a table $\text{Inc}[\cdot, \cdot]$ (see Lines 10, 18 and 20). Then, we plug this factor into the next lemma which provides our postprocessing.

Lemma 9 (∗). Let $s$ be a vertex and let $f : V(G)^2 \to \mathbb{N}$ be a function such that for each $u, v \in V(G)$ the value $f(u, v)$ can be computed in $O(\tau)$ time. Then, one can compute $\sum_{t \in V(G)} f(s, t) \cdot \sigma_{st}(v)$ for all $v \in V$ overall in $O(n \cdot \tau + m)$ time.

Our strategy is to start the algorithm behind Lemma 9 only from vertices in $V^{\geq 3}(G)$ (see Line 22). Since the term $\tau$ in the above lemma will be constant, we obtain a running time of $O(kn)$ for running this postprocessing for all vertices. The most intricate part will be to precompute the factors in $\text{Inc}[\cdot, \cdot]$ (see Lines 18 and 20). We defer the details to Section 3.1. In these parts, we need the tables $W^{\text{left}}$ and $W^{\text{right}}$. These tables store values depending on the maximal induced path a vertex is in. More precisely, for a vertex $x_k$ in a maximal induced path $p_{\text{max}} = x_0, x_1, \ldots, x_k$, we store in $W^{\text{left}}[x_k]$ the sum of the $\text{Pen}[\cdot]$-values of vertices “left of” $x_k$ in $p_{\text{max}}$; formally, $W^{\text{left}}[x_k] = \sum_{i=1}^{k} \text{Pen}[x_i]$. Similarly, we have $W^{\text{right}}[x_k] = \sum_{i=k+1}^{n} \text{Pen}[x_i]$. The reason for having these tables is easy to see: Assume for the vertex $x_k \in p_{\text{max}}$ that the shortest paths to $t \notin V(p_{\text{max}})$ leave $p_{\text{max}}$ through $x_k$. Then, it is equivalent to just consider the shortest path(s) starting in $x_0$ and simulate the vertices between $x_k$ and $x_0$ in $p_{\text{max}}$ by “temporarily increasing” $\text{Pen}[x_0]$ by $W^{\text{left}}[x_k]$. This is also the idea behind the argument that we only need to increase the values $\text{Inc}[\cdot, \cdot]$ for the endpoints of the maximal induced paths in Line 18.

This leaves us with the remaining part of the preprocessing: the computation of the distances $d_G(s, t)$, the number of shortest paths $\sigma_{st}$, and $\text{Inc}[s, t]$ for $s \in V^{\geq 3}(G), t \in V(G)$ (see Lines 7 to 10 in Algorithm 1). This can be done in $O(kn)$ time as well:

Lemma 10 (∗). The initialization in the for-loop in Lines 7 to 10 of Algorithm 1 can be done in $O(kn)$ time.

Putting all parts together, we arrive at our main theorem (see Section 3.2 for the proof).

Theorem 11. Betweenness Centrality can be solved in $O(kn)$ time, where $k$ is the feedback edge number of the input graph.
3 Dealing with maximal induced paths

In this section, we focus on degree-two vertices contained in maximal induced paths. Recall that the goal is to compute the betweenness centrality \( C_B(v) \) (see Equation (1)) for all \( v \in V(G) \) in \( O(kn) \) time. In the end of this section, we finally prove Theorem 11.

Based on Observation 8 and Equation (1), we compute \( C_B(v) \) in three steps. By starting a modified BFS from vertices in \( V \geq 3(G) \) similarly to Baglioni et al. [2] and Brandes [3], we can compute the following term in \( O(kn) \) time:

\[
\sum_{s \in V \geq 3(G), t \in V(G)} \gamma(t, s, v) + \sum_{s \in V = 2(G), t \in V \geq 3(G)} \gamma(s, t, v).
\]

3.1 Paths with endpoints in maximal induced paths

In this subsection, we show how to compute the remaining two summands given in Observation 8. In the next subsection, we prove Theorem 11.

Paths with endpoints in different maximal induced paths. We now focus on shortest paths between pairs of maximal induced paths \( P_{1}^{\text{max}} \) and \( P_{2}^{\text{max}} \), and how to efficiently determine how these paths affect the betweenness centrality of each vertex.

▶ Proposition 12 (⋆). In \( O(kn) \) time the following values can be computed for all \( v \in V(G) \):

\[
\sum_{s \in V = 2(P_{1}^{\text{max}}), t \in V = 2(P_{2}^{\text{max}})} \gamma(s, t, v).
\]

Recall that in the course of the algorithm, we first gather values in \( \text{Inc[}[,\cdot]\} \) and in the final step we compute for each \( s, t \in V \geq 3(G) \) the values \( \text{Inc}[s,t] \cdot \sigma_{st}(v) \) in \( O(m) \) time (Lemma 9). This postprocessing (see Lines 21 and 22 in Algorithm 1) takes \( O(kn) \) time.

In the proof of Proposition 12 (deferred to the full version), we consider two cases for every pair \( P_{1}^{\text{max}} \neq P_{2}^{\text{max}} \in P_{\text{max}} \) of maximal induced paths: First, we look at how the shortest paths between vertices in \( P_{1}^{\text{max}} \) and \( P_{2}^{\text{max}} \) affect the betweenness centrality of those vertices that are not contained in the two maximal induced paths, and second, how they affect the betweenness centrality of those vertices that are contained in the two maximal induced paths.

Paths with endpoints in the same maximal induced paths. Subsequently, we look at shortest paths starting and ending in a maximal induced path \( P_{\text{max}} = x_{0} \ldots x_{q} \) and show how to efficiently compute how these paths affect the betweenness centrality. Our goal is to prove the following:

▶ Proposition 13. In \( O(kn) \) time the following values can be computed for all \( v \in V(G) \):

\[
\sum_{s, t \in V \geq 2(P_{\text{max}}), P_{\text{max}} \neq P_{\text{max}} \in P_{\text{max}}} \gamma(s, t, v).
\]

As in Section 3.1, we first gather all increments to \( \text{Inc[}[,\cdot]\} \) and in the final step, we compute for each \( s, t \in V \geq 3(G) \) the values \( \text{Inc}[s,t] \cdot \sigma_{st}(v) \). We start with the following simple observation.
Observation 14. Let $P^{\text{max}} = x_0 \ldots x_q$, where $x_0, x_q \in V^\geq 3(G)$ and $x_i \in V^= 2(G)$ for $1 \leq i \leq q - 1$. Then

$$\sum_{s,t \in V^{=2}(P^{\text{max}})} \gamma(s, t, v) = \sum_{i,j \in [1, q-1]} \gamma(x_i, x_j, v) = 2 \cdot \sum_{i=1}^{q-1} \sum_{j=i+1}^{q-1} \gamma(x_i, x_j, v).$$

For the sake of readability we set $[x_p, x_q] := \{x_p, x_{p+1}, \ldots, x_q\}$, $p < r$. We will distinguish between two different cases that we then treat separately: Either $v \in [x_i, x_j]$ or $v \in V(G) \setminus [x_i, x_j]$. We will show that both cases can be solved in overall $O(q)$ time for $P^{\text{max}}$. Doing this for all maximal induced paths results in a running time of $O(\sum_{P^{\text{max}} \subseteq P^{\text{max}}} V^=2(P^{\text{max}})) \subseteq O(n)$.

We will distinguish between the two main cases in the calculations — all shortest $x_i x_j$-paths are fully contained in $P^{\text{max}}$, or all shortest $x_i x_j$-paths leave $P^{\text{max}}$ — and the corner case that there are some shortest paths inside $P^{\text{max}}$ and some that partially leave it. Observe that for any fixed pair $i < j$ the distance between $x_i$ and $x_j$ is given by $d_{in} = j - i$ if a shortest path is contained in $P^{\text{max}}$ and by $d_{out} = i + d_G(x_0, x_q) + q - j$ if a shortest $x_i x_j$-path leaves $P^{\text{max}}$. The corner case that there are shortest paths both inside and outside of $P^{\text{max}}$ occurs when $d_{in} = d_{out}$. In this case it holds that $j - i = i + d_G(x_0, x_q) + q - j$ which is equivalent to

$$j = i + \frac{d_G(x_0, x_q) + q}{2},$$

where $j$ is an integer smaller than $q$. For convenience, we will use a notion of “mid-elements” for a fixed starting vertex $x_i$. We distinguish between the two cases that this mid-element has a higher index in $P^{\text{max}}$ or a lower one. Formally, we say that $i_{\text{mid}} = i + (d_G(x_0, x_q) + q)/2$ and $j_{\text{mid}} = j - (d_G(x_0, x_q) + q)/2$. We next analyze the factor $\sigma_{x_i, x_j}(v)/\sigma_{x_i, x_j}$. We also distinguish between the cases $v \in V(P^{\text{max}})$ and $v \notin V(P^{\text{max}})$. Observe that

$$\sigma_{x_i, x_j}(v) = \begin{cases} 0, & \text{if } d_{out} < d_{in} \land v \in [x_i, x_j] \lor d_{in} < d_{out} \land v \notin [x_i, x_j]; \\ 1, & \text{if } d_{in} < d_{out} \land v \in [x_i, x_j]; \\ 1, & \text{if } d_{in} < d_{out} \land v \notin [x_i, x_j] \land v \in V(P^{\text{max}}); \\ \frac{\sigma_{x_0, x_q}(v)}{\sigma_{x_0, x_q} + 1}, & \text{if } d_{in} = d_{out} \land v \notin [x_i, x_j]; \\ \frac{\sigma_{x_0, x_q}}{\sigma_{x_0, x_q} + 1}, & \text{if } d_{in} = d_{out} \land v \notin [x_i, x_j] \land v \in V(P^{\text{max}}); \\ \frac{\sigma_{x_0, x_q}(v)}{\sigma_{x_0, x_q} + 1}, & \text{if } d_{in} = d_{out} \land v \notin V(P^{\text{max}}). \end{cases}$$

The denominator $\sigma_{x_0, x_q} + 1$ is correct since there are $\sigma_{x_0, x_q}$ shortest paths from $x_0$ to $x_q$ (and therefore $\sigma_{x_0, x_q}$ shortest paths from $x_i$ to $x_j$ that leave $P^{\text{max}}$) and one shortest path from $x_i$ to $x_j$ within $P^{\text{max}}$. Note that if there are shortest paths that are not contained in $P^{\text{max}}$, then $d_G(x_0, x_q) < q$ as we are in the case that $0 < i < j < q$. Thus $P^{\text{max}}$ is not a shortest path from $x_0$ to $x_q$.

We will now compute the value for all paths that only consist of vertices in $P^{\text{max}}$, that is, we will compute for each $x_k$ with $i < k < j$ the term $2 \cdot \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \gamma(x_i, x_j, x_k)$ with a dynamic program in $O(q)$ time. Since $i < k < j$ this can be simplified to

$$2 \cdot \sum_{i \in [1, q-1]} \sum_{j \in [i+1, q-1]} \gamma(x_i, x_j, x_k) = 2 \cdot \sum_{i \in [1, k-1]} \sum_{j \in [k+1, q-1]} \gamma(x_i, x_j, x_k).$$

Lemma 15. For a fixed maximal induced path $P^{\text{max}} = x_0 x_1 \ldots x_q$, for all $x_k$ with $0 \leq k \leq q$ we can compute $2 \cdot \sum_{i \in [1, k-1]} \sum_{j \in [k+1, q-1]} \gamma(x_i, x_j, x_k)$ in $O(q)$ time.
An Adaptive Version of Brandes’ Algorithm for Betweenness Centrality

Proof. For the sake of readability we define

\[ \alpha_{x_k} = 2 \cdot \sum_{i \in [1, k-1]} \sum_{j \in [k+1, q-1]} \gamma(x_i, x_j, x_k). \]

Note that \( i \geq 1 \) and \( k > i \) and thus for \( x_0 \) we have \( \alpha_{x_0} = 2 \sum_{i \in \emptyset} \sum_{j \in [1, q-1]} \gamma(x_i, x_j, x_0) = 0. \) This will be the base case of the dynamic program.

For every vertex \( x_k \) with \( 1 \leq k < q \) it holds that

\[ \alpha_{x_k} = 2 \cdot \sum_{i \in [1, k-1]} \sum_{j \in [k+1, q-1]} \gamma(x_i, x_j, x_k) = 2 \cdot \sum_{i \in [1, k-2]} \sum_{j \in [k+1, q-1]} \gamma(x_i, x_j, x_k) + 2 \cdot \sum_{i \in [1, k-2]} \sum_{j \in [k+1, q-1]} \gamma(x_{k-1}, x_j, x_k). \]

Similarly, for \( x_k \) with \( 1 < k \leq q \) it holds that

\[ \alpha_{x_{k-1}} = 2 \cdot \sum_{i \in [1, k-2]} \sum_{j \in [k, q-1]} \gamma(x_i, x_j, x_{k-1}) = 2 \cdot \sum_{i \in [1, k-2]} \sum_{j \in [k, q-1]} \gamma(x_i, x_j, x_{k-1}) + 2 \cdot \sum_{i \in [1, k-2]} \sum_{j \in [k, q-1]} \gamma(x_i, x_k, x_{k-1}). \]

Next, observe that any path from \( x_i \) to \( x_j \) with \( i \leq k-2 \) and \( j \geq k+1 \) that contains \( x_k \) also contains \( x_{k-1} \) and vice versa. Substituting this into the equations above yields

\[ \alpha_{x_k} = \alpha_{x_{k-1}} + 2 \cdot \sum_{j \in [k+1, q-1]} \gamma(x_{k-1}, x_j, x_k) - 2 \cdot \sum_{i \in [1, k-2]} \gamma(x_i, x_k, x_{k-1}). \]

Lastly, we prove that \( \sum_{j \in [k+1, q-1]} \gamma(x_{k-1}, x_j, x_k) \) and \( 2 \cdot \sum_{i \in [1, k-2]} \gamma(x_i, x_k, x_{k-1}) \) can be computed in constant time once \( W^{\text{left}} \) and \( W^{\text{right}} \) are precomputed (see Lines 11 to 14 in Algorithm 1). These tables can be computed in \( O(q) \) time as well. For convenience we say that \( \gamma(x_i, x_j, x_k) = 0 \) if \( i \) or \( j \) are not integral or are not in \([1, q-1]\) and define \( W[x_i, x_j] = \sum_{\ell \in \text{Pen}[x_i]} W^{\text{left}}[x_j] - W^{\text{left}}[x_{\ell}]. \)

Then we can use Equations (2) and (3) to show that

\[ \sum_{j \in [k+1, q-1]} \gamma(x_{k-1}, x_j, x_k) = \sum_{j \in [k+1, q-1]} \text{Pen}[x_{k-1}] \cdot \text{Pen}[x_j] \cdot \frac{\sigma_{x_{k-1}x_j}(x_k)}{\sigma_{x_{k-1}x_k}}. \]

\[ = \gamma(x_{k-1}, x_{(k-1)_{\text{mid}}^+}, x_k) + \sum_{j \in [k+1, \min\{(k-1)_{\text{mid}}^+], 1, q-1\}]} \text{Pen}[x_{k-1}] \cdot \text{Pen}[x_j]. \]

\[ = \begin{cases} \text{Pen}[x_{k-1}] \cdot W[x_{k+1}, x_{q-1}], & \text{if } (k-1)_{\text{mid}}^+ \geq q; \\ \text{Pen}[x_{k-1}] \cdot W[x_{k+1}, x_{(k-1)_{\text{mid}}^+}], & \text{if } (k-1)_{\text{mid}}^+ < q \land (k-1)_{\text{mid}}^+ \notin \mathbb{Z}; \\ \text{Pen}[x_{k-1}] \cdot (\text{Pen}[x_{(k-1)_{\text{mid}}^+}] \cdot \frac{1}{\sigma_{x_{(k-1)_{\text{mid}}^+}}^x} + W[x_{k+1}, x_{(k-1)_{\text{mid}}^+}]), & \text{otherwise.} \end{cases} \]

Herein we use \((k-1)_{\text{mid}}^+ \notin \mathbb{Z}\) to say that \((k-1)_{\text{mid}}^+\) is not integral. Analogously,

\[ \sum_{i \in [1, k-2]} \gamma(x_i, x_k, x_{k-1}) = \sum_{i \in [1, k-2]} \text{Pen}[x_i] \cdot \text{Pen}[x_k] \cdot \frac{\sigma_{x_ix_k}(x_{k-1})}{\sigma_{x_ix_k}}. \]

\[ = \gamma(x_{k-1}, x_{k_{\text{mid}^-}}, x_k) + \sum_{i \in [\max\{1, (k-1)_{\text{mid}^-}+1\}, k-2]} \text{Pen}[x_i] \cdot \text{Pen}[x_k]. \]

\[ = \begin{cases} \text{Pen}[x_k] \cdot W[x_1, x_{k-2}], & \text{if } k_{\text{mid}^-} < 1; \\ \text{Pen}[x_k] \cdot W[x_{k_{\text{mid}^-}+1}, x_{k-2}], & \text{if } k_{\text{mid}^-} \geq 1 \land k_{\text{mid}^-} \notin \mathbb{Z}; \\ \text{Pen}[x_k] \cdot (\text{Pen}[x_{k_{\text{mid}^-}}] \cdot \frac{1}{\sigma_{x_{k_{\text{mid}^-}}^x}} + W[x_1, x_{k_{\text{mid}^-}+1}]), & \text{otherwise.} \end{cases} \]

This completes the proof since \((k-1)_{\text{mid}^+}, k_{\text{mid}^-}\), every entry in \(W[\cdot]\), and all other variables in the equation above can be computed in constant time once \(W^{\text{left}}[\cdot]\) is computed. Thus, computing \(\alpha_{x_k}\) for each vertex \(x_i\) in \(P^{\max}\) takes constant time. As there are \(q\) vertices in \(P^{\max}\), the computations for the whole maximal induced path \(P^{\max}\) take \(O(q)\) time. \(\square\)
We still need to compute the value for all paths that partially leave $P_{\text{max}}$. Note that $\text{Inc}[s, t] \cdot \sigma_{st}(v)$ will be computed in the postprocessing step (see Lines 21 and 22 in Algorithm 1).

**Lemma 16** ($\star$). Let $P_{\text{max}} = x_0 x_1 \ldots x_q \in P_{\text{max}}$. Then, assuming that $\text{Inc}[s, t] \cdot \sigma_{st}(v)$ can be computed in constant time for some $s, t \in V^{\geq 3}(G)$, for $v \in V(G) \setminus [x_i, x_j]$ one can compute $\sum_{i \in [1, q-1]} \sum_{j \in [i+1, q-1]} \gamma(x_i, x_j, v)$ in $O(q)$ time.

### 3.2 Postprocessing and algorithm summary

We are now ready to combine all parts and prove our main theorem. 

**Theorem 11** (Restated). Betweenness Centrality can be solved in $O(kn)$ time, where $k$ is the feedback edge number of the input graph.

**Proof.** We show that in the Lines 7 to 22 Algorithm 1 computes the value 

$$C_B(v) = \sum_{s, t \in V(G)} \text{Pen}[s] \cdot \text{Pen}[t] \cdot \frac{\sigma_{st}(v)}{\sigma_{st}} = \sum_{s, t \in V(G)} \gamma(s, t, v)$$

for all $v \in V(G)$ in $O(kn)$ time. We use Observation 8 to split the sum as follows:

$$\sum_{s, t \in V(G)} \gamma(s, t, v) = \sum_{s \in V^{\geq 3}(G), t \in V(G)} \gamma(s, t, v) + \sum_{s \in V^{= 2}(G), t \in V^{\geq 3}(G)} \gamma(t, s, v)$$

$$+ \sum_{s \in V^{= 2}(P_{\text{max}}), t \in V^{= 2}(P_{\text{max}})} \gamma(s, t, v) + \sum_{s, t \in V^{= 2}(P_{\text{max}})} \gamma(s, t, v).$$

By Propositions 12 and 13, we can compute the third and fourth summand in $O(kn)$ time provided that $\text{Inc}[s, t] \cdot \sigma_{st}(v)$ is computed for every $s, t \in V^{\geq 3}(G)$ and $v \in V(G)$ in a postprocessing step (see Lines 15 to 20). We incorporate this postprocessing into the computation of the first two summands in the equation, that is, we next show that for all $v \in V(G)$ the following value can be computed in $O(kn)$ time:

$$\sum_{s \in V^{\geq 3}(G), t \in V(G)} \gamma(s, t, v) + \sum_{s \in V^{\geq 3}(G), t \in V^{\geq 3}(G)} \gamma(s, t, v) + \sum_{s \in V^{= 2}(G), t \in V^{= 2}(G)} \text{Inc}[s, t] \cdot \sigma_{st}(v).$$

To this end, observe that

$$\sum_{s \in V^{\geq 3}(G), t \in V(G)} \gamma(s, t, v) + \sum_{s \in V^{\geq 3}(G), t \in V^{\geq 3}(G)} \gamma(s, t, v) + \sum_{s \in V^{\geq 3}(G), t \in V^{\geq 3}(G)} \text{Inc}[s, t] \cdot \sigma_{st}(v)$$

$$= \sum_{s \in V^{\geq 3}(G), t \in V(G)} \text{Pen}[s] \cdot \text{Pen}[t] \cdot \frac{\sigma_{st}(v)}{\sigma_{st}} + \sum_{s \in V^{\geq 3}(G), t \in V^{\geq 3}(G)} \text{Pen}[s] \cdot \text{Pen}[t] \cdot \frac{\sigma_{st}(v)}{\sigma_{st}} + \sum_{s \in V^{\geq 3}(G), t \in V^{\geq 3}(G)} \text{Inc}[s, t] \cdot \sigma_{st}(v)$$

$$= \sum_{s \in V^{\geq 3}(G)} \left(2 \cdot \sum_{t \in V^{\geq 2}(G)} \text{Pen}[s] \cdot \text{Pen}[t] \cdot \frac{\sigma_{st}(v)}{\sigma_{st}} + \sum_{t \in V^{\geq 3}} \sigma_{st}(v) \left(\frac{\text{Pen}[s] \cdot \text{Pen}[t]}{\sigma_{st}} + \text{Inc}[s, t]\right)\right).$$

Note that we initialize $\text{Inc}[s, t]$ in Lines 10 and 9 in Algorithm 1 with $2 \cdot \text{Pen}[s] \cdot \text{Pen}[t]/\sigma_{st}$ and $\text{Pen}[s] \cdot \text{Pen}[t]/\sigma_{st}$ respectively. Thus we can use the algorithm described in Lemma 9 for each vertex $s \in V^{\geq 3}(G)$ with $f(s, t) = \text{Inc}[s, t]$. 

ISAAC 2018
Since the values $Pen[s]$, $Pen[t]$, $\sigma_{st}$ and $Inc[s,t]$ can all be looked up in constant time, the algorithm takes $O(n + m)$ time to run a modified BFS from some vertex $s$ (see Lines 21 and 22). By Lemma 5 there are $O(\min\{k, n\})$ vertices of degree at least three. The algorithm therefore take $O(\min\{n, k\} \cdot m) = O(\min\{n, k\} \cdot (n + k)) = O(kn)$ time to run the modified BFS from all vertices of degree at least three.

4 Conclusion

Lifting the processing of degree-one vertices due to Baglioni et al. [2, 13] to a technically much more demanding processing of degree-two vertices, we derived a new algorithm for Betweenness Centrality running in $O(kn)$ worst-case time ($k$ is the feedback edge number of the input graph). Our work focuses on algorithm theory and contributes to the field of adaptive algorithm design [5] as well as to the recent “FPT in P” program [7]. It would be of high interest to identify structural parameterizations “beyond” the feedback edge number that might help to get more results in the spirit of our work. In particular, extending our algorithmic approach with the treatment of twin vertices [12, 13] might help to get a running time bound involving the vertex cover number of the graph. From a practical viewpoint it remains to be investigated for which classes of real-world networks our (more complicated) algorithmic approach yields faster algorithms in empirical studies.

References


Algorithms for Coloring Reconfiguration Under Recolorability Constraints

Hiroki Osawa
Graduate School of Information Sciences, Tohoku University, Japan
osawa@ecei.tohoku.ac.jp

Akira Suzuki
Graduate School of Information Sciences, Tohoku University, Japan
a.suzuki@ecei.tohoku.ac.jp
https://orcid.org/0000-0002-5212-0202

Takehiro Ito
Graduate School of Information Sciences, Tohoku University, Japan
takehiro@ecei.tohoku.ac.jp
https://orcid.org/0000-0002-9912-6898

Xiao Zhou
Graduate School of Information Sciences, Tohoku University, Japan
zhou@ecei.tohoku.ac.jp

Abstract

COLORING RECONFIGURATION is one of the most well-studied reconfiguration problems. In the problem, we are given two (vertex-)colorings of a graph using at most $k$ colors, and asked to determine whether there exists a transformation between them by recoloring only a single vertex at a time, while maintaining a $k$-coloring throughout. It is known that this problem is solvable in linear time for any graph if $k \leq 3$, while is PSPACE-complete for a fixed $k \geq 4$. In this paper, we further investigate the problem from the viewpoint of recolorability constraints, which forbid some pairs of colors to be recolored directly. More specifically, the recolorability constraint is given in terms of an undirected graph $R$ such that each node in $R$ corresponds to a color, and each edge in $R$ represents a pair of colors that can be recolored directly. In this paper, we give a linear-time algorithm to solve the problem under such a recolorability constraint if $R$ is of maximum degree at most two. In addition, we show that the minimum number of recoloring steps required for a desired transformation can be computed in linear time for a yes-instance. We note that our results generalize the known positive ones for COLORING RECONFIGURATION.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases combinatorial reconfiguration, graph algorithm, graph coloring

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.37

1 Partially supported by JST CREST Grant Number JPMJCR1402, and JSPS KAKENHI Grant Numbers JP17K12636 and JP18H04091, Japan.
2 Partially supported by JST CREST Grant Number JPMJCR1402, and JSPS KAKENHI Grant Numbers JP16K00004 and JP18H04091, Japan.
3 Partially supported by JSPS KAKENHI Grant Number JP16K00003, Japan.

© Hiroki Osawa, Akira Suzuki, Takehiro Ito, and Xiao Zhou; licensed under Creative Commons License CC-BY
 Algorithms for Coloring Reconfiguration Under Recolorability Constraints

1 Introduction

Combinatorial reconfiguration [10, 11, 13] has been studied intensively in the field of theoretical computer science. In a typical reconfiguration problem, we are given two feasible solutions of a search problem instance (e.g., graph colorings, independent sets, satisfying truth assignments), and asked to check the existence of a step-by-step transformation between them such that all intermediate results are also feasible and each step conforms to a fixed reconfiguration rule, that is, an adjacency relation defined on feasible solutions of the original search problem instance.

For example, the COLORING RECONFIGURATION problem is one of the most well-studied reconfiguration problems, defined as follows [3, 7]. For an integer $k \geq 1$, we are given two $k$-colorings $f_0$ and $f_r$ of the same graph $G$, and asked to determine whether there exists a sequence $\langle f_0, f_1, \ldots, f_\ell \rangle$ of $k$-colorings of $G$ such that $f_\ell = f_r$ and $f_i$ is obtained from $f_{i-1}$ by recoloring a single vertex of $G$ for each $i \in \{1, 2, \ldots, \ell\}$. Figure 1(c) shows an example of a desired sequence $\langle f_0, f_1, \ldots, f_\ell \rangle$ of 4-colorings, where $G$ is a complete graph $K_3$ as illustrated in Figure 1(a).

The complexity of COLORING RECONFIGURATION has been clarified based on several “standard” measures (e.g., the number of colors [3, 7, 12] and graph classes [1, 2, 5, 8, 9, 16]) which are used well also for analyzing the original search problem. On the other hand, in [14], we have introduced a new concept, called the recolorability constraint on colors, to analyze the complexity of COLORING RECONFIGURATION more precisely. This concept is newly tailored for COLORING RECONFIGURATION, and forbids some pairs of colors to be recolored directly.

1.1 Our problem

For an integer $k \geq 1$, let $C$ be the color set of $k$ colors $1, 2, \ldots, k$. Let $G$ be a graph with vertex set $V(G)$ and edge set $E(G)$. Recall that a $k$-coloring of $G$ is a mapping $f : V(G) \to C$ such that $f(v) \neq f(w)$ holds for any edge $vw \in E(G)$. The recolorability on $C$ is given in terms of an undirected graph $R$, called the recolorability graph on $C$, such that $V(R) = C$; each edge $ij \in E(R)$ represents a “recolorable” pair of colors $i,j \in V(R) = C$. Then, two $k$-colorings $f$ and $f'$ of $G$ are adjacent (under $R$) if the following two conditions hold:

(a) $|\{v \in V(G) : f(v) \neq f'(v)\}| = 1$, that is, $f'$ can be obtained from $f$ by recoloring a single vertex $v \in V(G)$; and

(b) if $f(v) \neq f'(v)$ for a vertex $v \in V(G)$, then $f(v)f'(v) \in E(R)$, that is, the colors $f(v)$ and $f'(v)$ form a recolorable pair.

![Figure 1](image)
We also note that Johnson et al. [12] gave a linear-time algorithm to solve coloring reconfiguration. As we mentioned, understanding of the computational hardness of coloring reconfiguration requires only Condition (a) above, that is, we can recolor a vertex from any color to any color directly. Observe that this corresponds to the case where \( R \) is a complete graph of size \( k \), and hence our adjacency relation generalizes the known one.

Given a graph \( G \), two \( k \)-colorings \( f_0 \) and \( f_r \) of \( G \), and a recolorability graph \( R \) on \( C \), the coloring reconfiguration problem under recolorability is the decision problem of determining whether there exists a sequence \( (f_0, f_1, \ldots, f_\ell) \) of \( k \)-colorings of \( G \) such that \( f_i = f_r \) and \( f_i \) are adjacent under \( R \) for all \( i \in \{1, 2, \ldots, \ell\} \); such a desired sequence is called an \( (f_0 \rightarrow f_r) \)-reconfiguration sequence, and its length (i.e., the number of recoloring steps) is defined as \( \ell \). For example, the sequence \( (f_0, f_1, \ldots, f_7) \) in Figure 1(c) is an \( (f_0 \rightarrow f_7) \)-reconfiguration sequence whose length is seven.

We emphasize that the concept of recolorability constraints changes the reachability of \( k \)-colorings drastically. For example, the \( (f_0 \rightarrow f_7) \)-reconfiguration sequence in Figure 1(c) is a shortest one between \( f_0 \) and \( f_7 \) under the recolorability graph \( R \) in Figure 1(b). However, in coloring reconfiguration (in other words, if \( R \) would be \( K_4 \) and would have the edge joining colors 1 and 3), we can recolor the (top) vertex of \( G \) from 1 to 3 directly. As another example, the instance illustrated in Figure 2 is a no-instance for our problem, but is a yes-instance for coloring reconfiguration with \( k = 3 \).

1.2 Related and known results

As we mentioned, coloring reconfiguration has been studied intensively [1, 2, 3, 4, 5, 7, 8, 9, 12, 15, 16]. In particular, a sharp analysis has been obtained from the viewpoint of the number \( k \) of colors: Bonsma and Cereceda [3] proved that coloring reconfiguration is \( \text{PSPACE} \)-complete even for a fixed \( k \geq 4 \). On the other hand, Cereceda et al. [7] proved that coloring reconfiguration is solvable in polynomial time for any graph if \( k \in \{1, 2, 3\} \). Brewster et al. [6] generalized this sharp analysis to circular coloring reconfiguration. We also note that Johnson et al. [12] gave a linear-time algorithm to solve coloring reconfiguration for any graph and \( k \in \{1, 2, 3\} \); indeed, their algorithm can determine in linear time whether an \((f_0 \rightarrow f_r)\)-reconfiguration sequence exists or not, and can compute its shortest length in linear time if it exists.

In [14], we introduced the concept of recolorability constraints, and showed the computational hardness of coloring reconfiguration under recolorability based on the graph structure of recolorability graphs \( R \). More specifically, we proved that the problem is \( \text{PSPACE} \)-complete if (1) \( R \) is of maximum degree at least four, or (2) \( R \) contains a connected component having at least two cycles. These results are strong in the sense that they show the \( \text{PSPACE} \)-completeness for all recolorability graphs satisfying (1) or (2). Furthermore, the latter result (2) implies that the problem is \( \text{PSPACE} \)-complete if \( R = K_4 \). Therefore, the results (1) and (2) generalize the known \( \text{PSPACE} \)-completeness for coloring reconfiguration with \( k \geq 4 \). In this sense, the results in [14] gave a sharper analysis and a better understanding of the computational hardness of coloring reconfiguration.

![Figure 2](image-url)
1.3 Our contribution

Despite the concept of recolorability graphs \( R \) generalized and sharpened the known PSPACE-completeness successfully, there is no algorithmic (positive) result for **COLORING RECONFIGURATION UNDER RECOLORABILITY** except for the special case of \( R = K_3 \) obtained from **COLORING RECONFIGURATION** [7, 12]. In this paper, we thus study the polynomial-time solvability of our problem, and generalize the known algorithmic results from the viewpoint of the graph structure of recolorability graphs. Specifically, our main result can be stated as the following theorem:

**Theorem 1.** Suppose that a recolorability graph \( R \) is of maximum degree at most two, and let \( k = |V(R)| \). For any graph \( G \) with \( n \) vertices and \( m \) edges, **COLORING RECONFIGURATION UNDER RECOLORABILITY** can be solved in \( O(k + n + m) \) time. Furthermore, if an \( (f_0 \to f_r) \)-reconfiguration sequence exists for two \( k \)-colorings \( f_0 \) and \( f_r \) of \( G \), then

- its shortest length can be computed in \( O(k + n + m) \) time; and
- a shortest \( (f_0 \to f_r) \)-reconfiguration sequence can be output in \( O(kn(n + m)) \) time.

We emphasize that Theorem 1 holds for any graph \( G \), and only the structure of \( R \) is restricted. Since \( K_3 \) is of maximum degree two, Theorem 1 generalizes the known positive results for **COLORING RECONFIGURATION** [7, 12]. Note that \( k \) is not always a constant (indeed, can be larger than \( n \)).

In this paper, we prove Theorem 1 as follows. We start by giving an observation that a recolorability graph \( R \) can be assumed to be connected without loss of generality (Section 2). Then, since the maximum degree of \( R \) is two, \( R \) is either a path or a cycle. In Section 3, we will prove Theorem 1 for the case where \( R \) is a path. Sections 4 and 5 are devoted to the case where \( R \) is a cycle; the algorithm in Section 4 only checks whether a given instance is a yes-instance or not, and the one in Section 5 computes the shortest length for a yes-instance.

Due to the page limitation, proofs of the claims marked with (*) are omitted from this extended abstract.

2 Preliminaries

Since we deal with (vertex-)coloring, we may assume without loss of generality that an input graph \( G \) is simple, connected and undirected. Let \( n = |V(G)| \) and \( m = |E(G)| \). For a vertex subset \( V' \subseteq V(G) \), we denote by \( G[V'] \) the subgraph of \( G \) induced by \( V' \).

For a graph \( G \) and a recolorability graph \( R \) on \( G \), we define the \( R \)-reconfiguration graph on \( G \), denoted by \( C_R(G) \), as follows: \( C_R(G) \) is an undirected graph such that each node of \( C_R(G) \) corresponds to a \( k \)-coloring of \( G \), and two nodes in \( C_R(G) \) are joined by an edge if their corresponding \( k \)-colorings are adjacent under \( R \). We sometimes call a node in \( C_R(G) \) simply a \( k \)-coloring if it is clear from the context. A path in \( C_R(G) \) from a \( k \)-coloring \( f \) to another one \( f' \) is called an \( (f \to f') \)-reconfiguration sequence. Note that any \( (f \to f') \)-reconfiguration sequence is reversible, that is, the path in \( C_R(G) \) forms an \( (f' \to f) \)-reconfiguration sequence, too. Then, the **COLORING RECONFIGURATION** problem **UNDER RECOLORABILITY** can be seen as the decision problem of determining whether \( C_R(G) \) contains an \( (f_0 \to f_r) \)-reconfiguration sequence for two given \( k \)-colorings \( f_0 \) and \( f_r \) of \( G \). Note that the problem does not ask for an actual \( (f_0 \to f_r) \)-reconfiguration sequence as the output. We always denote by \( f_0 \) and \( f_r \) two given \( k \)-colorings of \( G \) as an input of the problem. For two \( k \)-colorings of \( f \) and \( f' \) in \( C_R(G) \), we denote by \( \text{dist}(f, f') \) the shortest length (i.e., the minimum number of edges in \( C_R(G) \)) of an \( (f \to f') \)-reconfiguration sequence if it exists; otherwise we let \( \text{dist}(f, f') = +\infty \).
We note that a given recolorability graph $R$ can be assumed to be connected without loss of generality. To see this, first observe that no $(f_0 \to f_r)$-reconfiguration sequence exists if there is a vertex $u \in V(G)$ such that the colors $f_0(u)$ and $f_r(u)$ belong to different connected components of $R$. Next, consider any two vertices $v, w \in V(G)$ such that the colors $f_0(v)$ and $f_0(w)$ belong to different connected components $R_1$ and $R_2$ of $R$, respectively. Then, since $V(R_1) \cap V(R_2) = \emptyset$, we can independently recolor vertices $v$ and $w$. In this way, we can assume without loss of generality that $R$ is connected.

To describe our algorithms, we sometimes use the notion of digraphs (i.e., directed graphs). For an undirected graph $G$, we denote by $\overrightarrow{G}$ a digraph whose underlying graph is $G$, and also denote by $A(\overrightarrow{G})$ the arc set of $\overrightarrow{G}$. We denote by $vw$ an edge joining two vertices $v$ and $w$ in an undirected graph, while by $(v, w)$ an arc from $v$ to $w$ in a digraph. In this paper, we say that a digraph $\overrightarrow{G}$ is connected if $\overrightarrow{G}$ is weakly connected, that is, the underlying graph $G$ is connected. A vertex $v$ in a digraph $\overrightarrow{G}$ is called a source vertex if the in-degree of $v$ is zero, while it is called a sink vertex if the out-degree of $v$ is zero. A sequence $v_0 a_1 v_1 a_2 v_2 \ldots a_l v_l$ of vertices $v_0, v_1, \ldots, v_l$ and arcs $a_1, a_2, \ldots, a_l$ in $\overrightarrow{G}$ is called a forward walk from $v_0$ on $\overrightarrow{G}$ if it forms a directed walk from $v_0$ to $v_l$ (with repeated arcs and vertices allowed), that is, $a_i$ is the arc from $v_{i-1}$ to $v_i$ for all $i \in \{1, 2, \ldots, l\}$; while it is called a backward walk to $v_0$ on $\overrightarrow{G}$ if it is a directed walk from $v_l$ to $v_0$, that is, $a_i$ is the arc from $v_i$ to $v_{i-1}$ for all $i \in \{l, l-1, \ldots, 1\}$.

## 3 Algorithms for Path Recolorability

In this section, we consider the case where $R$ is a path. We first prove that the existence of an $(f_0 \to f_r)$-reconfiguration sequence can be checked in linear time, as follows.

**Theorem 2.** Coloring reconfiguration under recolorability for any graph $G$ can be solved in $O(k + n + m)$ time if a recolorability graph $R$ is a path.

We prove Theorem 2 by giving such an algorithm. We first rename the colors in $R$ so that the colors $1, 2, \ldots, k$ appear in a numerical order along the path $R$, and modify two $k$-colorings $f_0$ and $f_r$ accordingly; this can be done in $O(k + n)$ time. Then, the most important property for the path recolorability is that any recoloring step preserves the “order” of colors assigned to two adjacent vertices in $G$: If a $k$-coloring $f$ of $G$ assigns colors to two adjacent vertices $v, w \in V(G)$ such that $f(v) < f(w)$, then $f'(v) < f'(w)$ holds for any $k$-coloring $f'$ such that an $(f \to f')$-reconfiguration sequence exists. Indeed, this property yields the following necessary and sufficient condition, which can be checked in $O(m)$ time; and hence Theorem 2 holds.

**Lemma 3 (⋆).** An $(f_0 \to f_r)$-reconfiguration sequence exists on $\mathcal{C}_R(G)$ if and only if $f_r(v) < f_r(w)$ holds for any $vw \in E(G)$ such that $f_0(v) < f_0(w)$.

We next give a linear-time algorithm to compute $\text{dist}(f_0, f_r)$; together with Theorem 2, this completes the proof of Theorem 1 for the path recolorability.

**Theorem 4.** Suppose that a recolorability graph $R$ is a path, and let $f_0$ and $f_r$ be two $k$-colorings of a graph $G$ such that an $(f_0 \to f_r)$-reconfiguration sequence exists on $\mathcal{C}_R(G)$. Then,

(a) $\text{dist}(f_0, f_r) = \sum_{v \in V(G)} |f_r(v) - f_0(v)|$;

(b) $\text{dist}(f_0, f_r)$ can be computed in $O(k + n + m)$ time; and

(c) a shortest $(f_0 \to f_r)$-reconfiguration sequence can be output in $O(kn(n + m))$ time.
By Theorem 2 we can check in $O(k + n + m)$ time if an $(f_0 \rightarrow f_r)$-reconfiguration sequence exists on $C_R(G)$. Then, Theorem 4(b) immediately follows from Theorem 4(a). Therefore, we will prove Theorem 4(a) and (c), as follows: Observe that $\text{dist}(f_0, f_r) \geq \sum_{v \in V(G)} |f_r(v) - f_0(v)|$ holds, because each recoloring step can change the current color of a vertex $v \in V(G)$ to its adjacent color in $R$, and hence each vertex $v \in V(G)$ requires at least $|f_r(v) - f_0(v)|$ recoloring steps. Therefore, the following lemma completes the proof of Theorem 4.

Lemma 5 (*). There exists an $(f_0 \rightarrow f_r)$-reconfiguration sequence on $C_R(G)$ of length $\sum_{v \in V(G)} |f_r(v) - f_0(v)|$. Furthermore, it can be output in $O(kn(n + m))$ time.

4 Algorithm for Reachability on Cycle Recolorability

In this section, we consider the case where $R$ is a cycle, and show that the existence of an $(f_0 \rightarrow f_r)$-reconfiguration sequence can be checked in linear time; the shortest length will be discussed in the next section. We prove the following theorem in this section.

Theorem 6. Coloring reconfiguration under recolorability for any graph $G$ can be solved in $O(k + n + m)$ time if a recolorability graph $R$ is a cycle.

Since $K_3$ is a cycle, Theorem 6 immediately implies the following corollary.

Corollary 7 ([12]). Coloring reconfiguration with $k = 3$ can be solved in linear time.

We will prove Theorem 6 by giving such an algorithm, as follows. In Section 4.1, we give a simple necessary condition for a yes-instance based on the concept of “frozen” vertices; the idea is simple, but we need a nice characterization of frozen vertices for checking the condition in linear time. In Section 4.2, we then give a necessary and sufficient condition for a yes-instance by defining a potential function which appropriately characterizes the reconfigurability of $k$-colorings; however, this condition cannot be checked in linear time by a naive way. In Section 4.3, we thus explain how to check the condition in linear time.

We rename the colors in $R$ so that the colors 1, 2, ..., $k$ appear in a numerical order along the cycle $R$, and modify two $k$-colorings $f_0$ and $f_r$ accordingly; this can be done in $O(k + n)$ time. For notational convenience, we define the successor color $c^+$ and the predecessor color $c^-$ for a color $c \in V(R)$, as follows:

$$c^+ = \begin{cases} c + 1 & \text{if } c < k; \\ 1 & \text{if } c = k, \end{cases} \quad \text{and} \quad c^- = \begin{cases} c - 1 & \text{if } c > 1; \\ k & \text{if } c = 1. \end{cases}$$

We use this notation also for a color assigned by a $k$-coloring: For a $k$-coloring $f$ of a graph $G$ and a vertex $v$ in $G$, we denote by $f(v)^+$ and $f(v)^-$ the successor and predecessor colors for $f(v)$, respectively. In this and later sections, we call a $k$-coloring of $G$ simply a coloring.

4.1 Frozen vertices

We now define the concept of “frozen” vertices [7] from the viewpoint of recoloring, which plays an important role in our algorithm. For a coloring $f$ of a graph $G$ and a recolorability graph $R$ on $C$, a vertex $v \in V(G)$ is said to be frozen on $f$ (under $R$) if $f(v) = f'(v)$ holds for any coloring $f'$ of $G$ such that $C_R(G)$ has an $(f \rightarrow f')$-reconfiguration sequence. For a coloring $f$ of $G$, we denote by Frozen$(f)$ the set of all vertices in $G$ that are frozen on $f$. The following lemma gives a simple necessary condition, which immediately follows from the definition of frozen vertices.
Let $H$ be any digraph whose underlying graph is a subgraph of $G$. For a coloring $f$ of $G$ and each arc $(u, v) \in A(H)$, we define the potential $p_f((u, v))$ of $(u, v)$ on $f$, as follows:

$$p_f((u, v)) = \begin{cases} f(v) - f(u) & \text{if } f(v) > f(u); \\ f(v) - f(u) + k & \text{if } f(v) < f(u). \end{cases}$$
Note that \( f(u) \neq f(v) \) holds since \( uv \in E(G) \). In addition, observe that
\[
\|f|(u, v)| + \|f|(v, u)| = k \tag{1}
\]
holds for any pair of parallel arcs \((u, v)\) and \((v, u)\) if such a pair exists. The potential \( \|f|\) of \( \overrightarrow{H} \) on \( f \) is defined to be the sum of potentials of all arcs of \( \overrightarrow{H} \) on \( f \), that is,
\[
\|f| \left( \overrightarrow{H} \right) = \sum_{(u, v) \in A(H)} \|f|(u, v)|.
\]

Let \( C \) be a cycle in an undirected graph \( G \). Then, there are only two possible orientations of \( C \) such that they form directed cycles, that is, either the clockwise direction or the anticlockwise direction; we always denote by \( \overrightarrow{C} \) and \( \overleftarrow{C} \) such the two possible orientations of \( C \). The following lemma immediately follows from Eq. (1).

\begin{lemma}
\begin{itemize}
\item Let \( f \) be a coloring of an undirected graph \( G \). Then, \( \|f| \left( \overrightarrow{C} \right) + \|f| \left( \overleftarrow{C} \right) = k|E(C)| \) for every cycle \( C \) in \( G \).
\end{itemize}
\end{lemma}

For a coloring \( f \) of an undirected graph \( G \), we define a supergraph \( G^f \) of \( G \) as follows:\footnote{We note that our construction of \( G^f \) is different from that by Cereceda et al. [7] so that the running time of our algorithm does not depend on \( k \).}

Let \( V(G^f) = V(G) \), and we arbitrarily add new edges between frozen vertices on \( G \) so that \( \text{Frozen}(f) \) induces a connected subgraph in the resulting graph. Then, since there are at most \( |V(G)| \) frozen vertices, \( G^f \) has \( |V(G)| \) vertices and at most \( |E(G)| + |V(G)| - 1 \) edges. Note that \( G^f = G \) if \( \text{Frozen}(f) = \emptyset \). Recall that two given colorings \( f_0 \) and \( f_r \) of \( G \) are assumed to satisfy \( \text{Frozen}(f_0) = \text{Frozen}(f_r) \) and \( f_0(v) = f_r(v) \) for every vertex \( v \) in \( \text{Frozen}(f_0) \). We can thus assume \( G^{f_0} = G^{f_r} \), and hence simply denote it by \( G^f \). Furthermore, since newly added edges join only frozen vertices, we have the following lemma.

\begin{lemma}
\begin{itemize}
\item There exists an \((f_0 \to f_r)\)-reconfiguration sequence on \( C_{R}(G) \) if and only if there exists an \((f_0 \to f_r)\)-reconfiguration sequence on \( C_{R}(G^f) \).
\end{itemize}
\end{lemma}

We are now ready to claim our necessary and sufficient condition, as follows.

\begin{theorem}
\begin{itemize}
\item Let \( f_0 \) and \( f_r \) be two colorings of a graph \( G \) such that \( \text{Frozen}(f_0) = \text{Frozen}(f_r) \), and \( f_0(v) = f_r(v) \) for all vertices \( v \in \text{Frozen}(f_0) \). Then, an \((f_0 \to f_r)\)-reconfiguration sequence exists on \( C_{R}(G) \) if and only if \( \|f_0| \left( \overrightarrow{C} \right) = \|f_r| \left( \overrightarrow{C} \right) \) holds for every cycle \( C \) in \( G^f \).
\end{itemize}
\end{theorem}

Before proving the theorem, we note that Theorem 13 is independent from the choice of the two orientations of a cycle \( C \), because Lemma 11 implies that \( \|f_0| \left( \overrightarrow{C} \right) = \|f_r| \left( \overrightarrow{C} \right) \) holds if and only if \( \|f_0| \left( \overleftarrow{C} \right) = \|f_r| \left( \overleftarrow{C} \right) \) holds. We also note that Theorem 13 does not directly yield a linear-time algorithm.

We first prove the only-if direction of Theorem 13. Suppose that there exists an \((f_0 \to f_r)\)-reconfiguration sequence on \( C_{R}(G) \). Then, Lemma 12 implies that \( C_{R}(G^f) \) contains an \((f_0 \to f_r)\)-reconfiguration sequence \( \langle f_0, f_1, \ldots, f_k \rangle \), where \( f_k = f_r \), and hence the only-if direction of Theorem 13 can be obtained from the following lemma.

\begin{lemma}[^{*}]
\begin{itemize}
\item Suppose that two colorings \( f \) and \( f' \) are adjacent on \( C_{R}(G^f) \). Then, \( \|f| \left( \overrightarrow{C} \right) = \|f'| \left( \overrightarrow{C} \right) \) holds for every cycle \( C \) in \( G^f \).
\end{itemize}
\end{lemma}

We then prove the if direction of Theorem 13: If \( \|f_0| \left( \overrightarrow{C} \right) = \|f_r| \left( \overrightarrow{C} \right) \) holds for every cycle \( C \) in \( G^f \), then an \((f_0 \to f_r)\)-reconfiguration sequence exists on \( C_{R}(G^f) \); Lemma 12 then implies that \( C_{R}(G) \) contains an \((f_0 \to f_r)\)-reconfiguration sequence.
Our proof is constructive, that is, we give an algorithm which indeed finds an \((f_0 \to f_r)\)-reconfiguration sequence on \(C_R(G^f)\). We say that a vertex \(v\) is fixed if it is colored with \(f_r(v)\) and our algorithm decides not to recolor \(v\) anymore. Thus, all frozen vertices are fixed. Our algorithm maintains the set of fixed vertices, denoted by \(F\). The following Algorithm 1 transforms \(f_0\) into a coloring \(f'_0\) of \(G^f\) so that \(F \neq \emptyset\), as the initialization.

### Algorithm 1 Initialization for Algorithm 2.

1. If \(\text{Frozen}(f_0) \neq \emptyset\), then let \(F = \text{Frozen}(f_0)\) and \(f'_0 = f_0\).
2. Otherwise let \(F = \{v\}\) for an arbitrarily chosen vertex \(v \in V(G)\). Let \(f = f_0\), and obtain \(f'_0\) such that \(f'_0(v) = f_r(v)\), as follows:
   2-1. If \(f(v) = f_r(v)\), then let \(f'_0 = f\) and stop the algorithm.
   2-2. Otherwise recolor a sink vertex \(w\) (possibly \(v\) itself) of \(B^+(v, f)\) to \(f(w)^+\). Let \(f\) be the resulting coloring, and go to Step 2-1.

Note that we can always find a sink vertex \(w\) in Step 2-2 of Algorithm 1, because otherwise \(\overrightarrow{B}^+(v, f)\) contains a directed cycle; by Lemma 9 the vertices in the directed cycle are frozen, and hence this contradicts the assumption that \(\text{Frozen}(f_0) = \emptyset\) holds in Step 2. We note the following properties.

**Lemma 15.** Let \(F \subseteq V(G^f)\) be the vertex subset obtained by Algorithm 1, and let \(f'_0\) be the coloring of \(G^f\) obtained by Algorithm 1. Then, the induced subgraph \(G^f[F]\) is connected, and \(p_{f_0}(\overrightarrow{C}) = p_{f_0}(\overrightarrow{C}) = p_{f_r}(\overrightarrow{C})\) for any cycle \(C\) in \(G^f\).

**Proof.** Recall that \(G^f\) was obtained by adding new edges to \(G\) so that \(G^f[\text{Frozen}(f_0)]\) is connected. Thus, \(G^f[F] = G^f[\text{Frozen}(f_0)]\) is connected if \(\text{Frozen}(f_0) \neq \emptyset\). If \(\text{Frozen}(f_0) = \emptyset\), then \(F\) consists of a single vertex \(v\); and hence \(G^f[F]\) is connected also in this case.

Notice that Algorithm 1 constructs an \((f_0 \to f'_0)\)-reconfiguration sequence on \(C_R(G^f)\). Then, Lemma 14 implies that \(p_{f_0}(\overrightarrow{C}) = p_{f_0}(\overrightarrow{C}) = p_{f_r}(\overrightarrow{C})\) for any cycle \(C\) in \(G^f\).

We now give our main procedure, called Algorithm 2, which finds an \((f'_0 \to f_r)\)-reconfiguration sequence on \(C_R(G^f)\). The algorithm attempts to extend the vertex set \(F\) to \(V(G^f)\) so that each vertex \(v\) in \(F\) is fixed (and hence is colored with \(f_r(v)\)); we eventually obtain the target coloring \(f_r\) when \(F = V(G^f)\). Recall that our algorithm never recolors any vertex \(v\) in \(F\), and all frozen vertices are contained in \(F\). Let \(f = f'_0\), and apply the following procedure.

### Algorithm 2 Finding an \((f'_0 \to f_r)\)-reconfiguration sequence on \(C_R(G^f)\).

1. If \(F = V(G^f)\) holds, then stop the algorithm.
2. Otherwise pick an arbitrary vertex \(v \in V(G^f) \setminus F\) which is adjacent with at least one vertex \(u \in F\).
   2-1. If \(f(v) = f_r(v)\), then add \(v\) to \(F\) and go to Step 1.
   2-2. Otherwise
      - if \(p_f((u, v)) < p_{f_r}((u, v))\), then recolor a sink vertex \(w\) (possibly \(v\) itself) of \(\overrightarrow{B}^+(v, f)\) to \(f(w)^+\); and
      - if \(p_f((u, v)) > p_{f_r}((u, v))\), then recolor a source vertex \(w\) (possibly \(v\) itself) of \(\overrightarrow{B}^-(v, f)\) to \(f(w)^-\).
    Let \(f\) be the resulting coloring, and go to Step 2-1.
To prove that Algorithm 2 correctly finds an \((f'_0 \to f_r)\)-reconfiguration sequence on \(\mathcal{C}_R(G^f)\), it suffices to show that there always exists a non-fixed sink/source vertex in Step 2-2 under the condition that \(p_{f_0}(\vec{C}) = p_{f_r}(\vec{C}) = p_{f_0}(\vec{C})\) holds for any cycle \(C\) in \(G^f\). Therefore, the following lemma completes the proof of the if direction of Theorem 13.

Lemma 16 (*). Every application of Step 2 of Algorithm 2 produces a set \(F\) of fixed vertices and a coloring \(f\) of \(G^f\) satisfying the following (a) and (b): For each edge \(uv\) in \(G^f\) such that \(u \in F\) and \(v \notin F\),

(a) if \(p_f((u, v)) < p_{f_r}((u, v))\), then \(\vec{B}^+(v, f)\) is a directed acyclic graph such that no vertex in \(\vec{B}^+(v, f)\) is contained in \(F\); and

(b) if \(p_f((u, v)) > p_{f_r}((u, v))\), then \(\vec{B}^-(v, f)\) is a directed acyclic graph such that no vertex in \(\vec{B}^-(v, f)\) is contained in \(F\).

4.3 Proof of Theorem 6

We finally prove Theorem 6 by giving such an algorithm. Our algorithm first checks the necessary and sufficient condition described in Lemma 8. By Lemma 10 this step can be done in \(O(m)\) time. Note that we can obtain the vertex subsets \(\text{Frozen}(f_0)\) and \(\text{Frozen}(f_r)\) in this running time. Then, we determine whether a given instance is a yes-instance or not, based on the necessary and sufficient condition described in Theorem 13. However, recall that the condition in Theorem 13 cannot be checked in linear time by a naive way. Below, we give a linear-time algorithm to check the condition.

Let \(T\) be an arbitrary spanning tree of the graph \(G^f\). For an edge \(e \in E(G^f) \setminus E(T)\), we denote by \(C_{T,e}\) the unique cycle obtained by adding the edge \(e\) to \(T\). The following lemma shows that it suffices to check the necessary and sufficient condition only for the number \(|E(G^f) \setminus E(T)|\) of cycles.

Lemma 17 (*). Let \(T\) be any spanning tree of \(G^f\). Then, \(p_{f_0}(\vec{C}) = p_{f_r}(\vec{C})\) holds for every cycle \(C\) of \(G^f\) if and only if \(p_{f_0}(\vec{C}_{T,e}) = p_{f_r}(\vec{C}_{T,e})\) holds for every edge \(e \in E(G^f) \setminus E(T)\).

Lemma 17 and the following lemma imply that there is a linear-time algorithm to check the necessary and sufficient condition described in Theorem 13. Therefore, the following lemma completes the proof of Theorem 6.

Lemma 18 (*). Let \(T\) be any spanning tree of \(G^f\). Then, \(p_{f_0}(\vec{C}_{T,e})\) and \(p_{f_r}(\vec{C}_{T,e})\) for all \(e \in E(G^f) \setminus E(T)\) can be computed in \(O(n + m)\) time in total.

5 Algorithm for Shortest Sequence on Cycle Recolorability

In this section, we consider the case where \(R\) is a cycle, and explain how to compute the length of a shortest reconfiguration sequence.

Let \(P_{u,v}\) be a path in an undirected graph \(G\) connecting vertices \(u\) and \(v\). We denote by \(\vec{P}_{u,v}\) the directed path from \(u\) to \(v\). The following theorem characterizes the shortest length of an \((f_0 \to f_r)\)-reconfiguration sequence, which generalizes the characterization for coloring reconfiguration with \(k = 3\) [7, 12].

Theorem 19 (*). Suppose that a recolorability graph \(R\) is a cycle, and let \(f_0\) and \(f_r\) be two colorings of a graph \(G\) such that an \((f_0 \to f_r)\)-reconfiguration sequence exists on \(\mathcal{C}_R(G)\). Then, the following (a) and (b) hold:
(a) If \( \text{Frozen}(f_0) \neq \emptyset \), then it holds for an arbitrary chosen vertex \( u \in \text{Frozen}(f_0) \) that

\[
\text{dist}(f_0, f_r) = \sum_{v \in V(G)} |p_{f_r}(P_{u,v}) - p_{f_0}(P_{u,v})|,
\]

where \( P_{u,v} \) is an arbitrary chosen path in \( G \) connecting \( u \) and \( v \).

(b) If \( \text{Frozen}(f_0) = \emptyset \), then there exist two integers \( \rho_{u,1} \) and \( \rho_{u,2} \) for an arbitrary chosen vertex \( u \in V(G) \) such that

\[
\text{dist}(f_0, f_r) = \min \left\{ \sum_{v \in V(G)} |p_{f_r}(P_{u,v}) - p_{f_0}(P_{u,v}) + \rho_{u,1}|, \sum_{v \in V(G)} |p_{f_r}(P_{u,v}) - p_{f_0}(P_{u,v}) + \rho_{u,2}| \right\},
\]

where \( P_{u,v} \) is an arbitrary chosen path in \( G \) connecting \( u \) and \( v \).

We finally claim that \( \text{dist}(f_0, f_r) \) can be computed in linear time, based on Theorem 19, and that a shortest \( (f_0 \rightarrow f_r) \)-reconfiguration sequence can be output in polynomial time.

\[\blacktriangleright \text{Lemma 20 (*)}. \text{For any vertex } u \in V(G), \text{two integers } \rho_{u,1} \text{ and } \rho_{u,2} \text{ of Theorem 19(b) can be obtained in } O(n + m) \text{ time. Furthermore,}
\]

(a) \( \text{dist}(f_0, f_r) \) can be computed in \( O(n + m) \) time; and

(b) a shortest \( (f_0 \rightarrow f_r) \)-reconfiguration sequence can be output in \( O(kn(n + m)) \) time.

6 Concluding Remarks

In this paper, we have generalized and sharpened the positive results [7, 12] obtained for \textsc{coloring reconfiguration}, from the viewpoint of recolorability constraints. We emphasize that our algorithms run in linear time to simply answer the decision problem \textsc{coloring reconfiguration under recolorability}, or to compute the shortest length of \( (f_0 \rightarrow f_r) \)-reconfiguration sequences.

One may expect that a shortest \( (f_0 \rightarrow f_r) \)-reconfiguration sequence can be output also in linear time. However, Cereceda et al. [7] showed that there exists an infinite family of yes-instances for \textsc{coloring reconfiguration} with \( k = 3 \) whose shortest \( (f_0 \rightarrow f_r) \)-reconfiguration sequence requires \( \Omega(n^2) \) length.

Together with our sister paper [14], we have clarified several tractable/intractable cases of \textsc{coloring reconfiguration under recolorability}. Our analyses are summarized in Table 1, and give a better understanding of the complexity of \textsc{coloring reconfiguration}. However, the complexity status remains open for the case where a connected recolorability graph \( R \) is of maximum degree three and has at most one cycle.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
Maximum degree of \( R \) & \( R \) contains at most one cycle & \( R \) contains at least two cycles \\
\hline
two & Linear time [this paper] & (no such \( R \) exists) \\
three & ? & PSPACE-complete [14] \\
\hline
\end{tabular}
\caption{Complexity of \textsc{coloring reconfiguration under recolorability}, where a recolorability graph \( R \) is assumed to be connected without loss of generality.}
\end{table}
References


A Cut Tree Representation for Pendant Pairs

On-Hei S. Lo
Institut für Mathematik, Technische Universität Ilmenau, Weimarer Strasse 25, D-98693 Ilmenau, Germany
solomon.lo@tu-ilmenau.de

Jens M. Schmidt
Institut für Mathematik, Technische Universität Ilmenau, Weimarer Strasse 25, D-98693 Ilmenau, Germany

Abstract
Two vertices $v$ and $w$ of a graph $G$ are called a pendant pair if the maximal number of edge-disjoint paths in $G$ between them is precisely $\min\{d(v), d(w)\}$, where $d$ denotes the degree function. The importance of pendant pairs stems from the fact that they are the key ingredient in one of the simplest and most widely used algorithms for the minimum cut problem today.

Mader showed 1974 that every simple graph with minimum degree $\delta$ contains $\Omega(\delta^2)$ pendant pairs; this is the best bound known so far. We improve this result by showing that every simple graph $G$ with minimum degree $\delta \geq 5$ or with edge-connectivity $\lambda \geq 4$ or with vertex-connectivity $\kappa \geq 3$ contains in fact $\Omega(\delta|V|)$ pendant pairs. We prove that this bound is tight from several perspectives, and that $\Omega(\delta|V|)$ pendant pairs can be computed efficiently, namely in linear time when a Gomory-Hu tree is given. Our method utilizes a new cut tree representation of graphs.

2012 ACM Subject Classification
Mathematics of computing → Graph theory, Mathematics of computing → Graph algorithms

Keywords and phrases
Pendant Pairs, Pendant Tree, Maximal Adjacency Ordering, Maximum Cardinality Search, Testing Edge-Connectivity, Gomory-Hu Tree

1 Introduction
The study of pendant pairs is motivated by the well-known, simple and widely used minimum cut algorithm of Nagamochi and Ibaraki [11], which refines the work of Mader [8, 7] in the early 70s, and was simplified by Stoer and Wagner [12] and Frank [3]. The key approach of this algorithm is to iteratively contract a pendant pair of the input graph in near-linear time by using maximal adjacency orderings (also known as maximum cardinality search [13]). Having done that $n - 1$ times, one can obtain a minimum cut by just considering the minimum degree of all intermediate graphs. In a break-through result, Kawarabayashi and Thorup [6] succeeded to give a near-linear time deterministic minimum cut algorithm for simple graphs, and this was later made faster by Henzinger et al. [4]. Hence, the algorithm of Nagamochi and Ibaraki is not the most efficient, but its simplicity is unmatched so far.

This motivates the following question: How many (distinct) pendant pairs does a graph with a given minimum degree possess? If there are many and, additionally, these could be computed efficiently, this would lead immediately to an improvement of the running time of...
the Nagamochi-Ibaraki algorithm. Here, we aim for the fundamental and natural question of finding a good lower bound on the number of distinct pendant pairs in graphs with a given minimum degree. We will mainly consider simple graphs, as these allow us to prove strong lower bounds (we give an example that shows that all bounds for multigraphs must be considerably weaker).

As early as 1973, and originally motivated by the structure of minimally $k$-edge-connected graphs, Mader proved that every graph with minimum degree $\delta \geq 1$ contains at least one pendant pair [8]. This holds also for the vertex-connectivity variant of pendant pairs, which nowadays is most easily proven by using maximal adjacency orderings. Later, Mader improved his result by showing that every simple graph with minimum degree $\delta$ contains $\Omega(\delta^2)$ pendant pairs [9].

Our main result in this paper sets the graph-theoretical prerequisite that the algorithmic approach described above of finding many pendant pairs might actually work out. We improve Mader’s result by showing that every simple graph that satisfies $\delta \geq 5$ or $\lambda \geq 4$ or $\kappa \geq 3$ contains $\Omega(\delta n)$ pendant pairs; this exhibits a dependency on $n := |V|$ instead of $\delta$, which is usually much larger. We prove that this result is tight with respect to the order of the bound and with respect to every assumption.

We show how to compute $\Omega(\delta n)$ pendant pairs from a Gomory-Hu tree in linear time. Clearly, computing a Gomory-Hu tree in advance does not match the best running time $O(m + n)$, $m := |E|$, for finding one pendant pair; however, we conjecture that it is actually possible to compute $\omega(1)$ pendant pairs in linear time. An affirmative answer to this would already imply a speed-up for the Nagamochi-Ibaraki-algorithm.

Our results utilize a new cut tree representation of graphs named pendant tree.

2 A Note on the History of Maximal Adjacency Orderings

Mader’s proof for the existence of one pendant pair relies strongly on [7, Lemma 1], which in turn uses special orderings on the vertices. Interestingly, these orderings are maximal adjacency orderings and this fact exhibits an apparently forgotten variant of them, which existed long before they got 1984 their first name (maximum cardinality search [13]).

We are only aware of one place in literature where this is (briefly) mentioned: [10, p. 443]. Mader’s existential proof can in fact be made algorithmic. A direct comparison between the old and the modern variant however shows that the modern maximal adjacency orderings are nicer to describe, as they work on the original graph, while Mader iteratively moves edges in the graph in order to represent the essential connectivity information on the already visited vertex set with a clique.

3 Preliminaries

All graphs considered in this paper are non-empty, finite, unweighted and undirected unless specified otherwise. Let $G := (V, E)$ be a graph. Contracting a vertex subset $X \subseteq V$ identifies all vertices in $X$ and deletes occurring self-loops (we do not require that $X$ induces a connected graph in $G$).

For non-empty and disjoint vertex subsets $X, Y \subset V$, let $E_G(X, Y)$ denote the set of all edges in $G$ that have one endvertex in $X$ and one endvertex in $Y$. Let further $\overline{X} := V - X$, $d_G(X, Y) := |E_G(X, Y)|$ and $d_G(X) := |E_G(X, \overline{X})|$; if $X = \{v\}$ for some vertex $v \in V$, we simply write $E_G(v, Y)$, $d_G(v, Y)$ and $d_G(v)$. A subset $\emptyset \neq X \subset V$ of a graph $G$ is called a cut of $G$. Let a cut $X$ of $G$ be trivial if $|X| = 1$ or $|\overline{X}| = 1$. Let the length and size of a path be the number of its edges and vertices, respectively. Let $\delta(G) := \min_{v \in V} d_G(v)$ be the minimum degree of $G$. For a vertex $v \in G$, let $N_G(v)$ be the set of neighbors of $v$ in $G$. 
For two vertices $v, w \in V$, let $\lambda_G(v, w)$ be the maximal number of edge-disjoint paths between $v$ and $w$ in $G$. A minimum $v$-$w$-cut is a cut $X$ that separates $v$ and $w$ and satisfies $\delta_G(X) = \lambda_G(v, w)$. Two vertices $v, w \in V$ are called $k$-edge-connected if $\lambda_G(v, w) \geq k$. The edge-connectivity $\lambda(G)$ of $G$ is the greatest integer such that every two distinct vertices are $\lambda(G)$-edge-connected. Let $\kappa(G)$ be the vertex-connectivity of $G$, i.e., the minimum number of vertices $U$ such that $G - U$ is disconnected. We omit parentheses for single elements (like vertices or edges) in set subtractions.

We call a pair $\{v, w\}$ of vertices pendant if $\lambda_G(v, w) = \min\{d_G(v), d_G(w)\}$. In order to increase readability, we will omit subscripts whenever the graph is clear from the context.

4 The Pendant Tree

We propose a new cut tree, which can be seen as a refinement of Gomory-Hu trees. The idea is to partition the vertex set such that each part consists only of vertices that are pairwise pendant, and impose a tree structure on these vertex subsets such that edges in this tree correspond to cuts in the graph that separate some non-pendant pair. For the sake of notational clarity, we will call the vertices of such trees blocks.

For a tree $T$ whose vertex set partitions $V$ and an edge $AB \in E(T)$, let $C_{AB}$ be the union of the blocks that are contained in the component of $T - AB$ containing $A$, and symmetrically, $C_{BA} = V - C_{AB}$. We will consider $T$ as a tree with edge weights as follows. For an edge $AB \in E(T)$, let $c(AB) := \delta_G(C_{AB})$ be the size of its corresponding edge-cut in $G$.

Definition 1. A non-pendant-pair covering tree, or simply pendant tree, $T$ of a graph $G = (V, E)$ is a tree whose vertex set partitions $V$ such that

(i) every two distinct vertices in a common block of this partition are pendant,
(ii) for every edge $AB \in E(T)$, there are vertices $a \in A$ and $b \in B$ such that $\{a, b\}$ is non-pendant, and
(iii) for every edge $AB \in E(T)$, there are vertices $a^* \in A$ and $b^* \in B$ such that $c(AB) = \lambda_G(a^*, b^*)$.

Note that $T$ is an auxiliary tree which is not obtained from $G$ by contracting vertex subsets. The following lemma allows us to find a non-pendant pair for every two adjacent blocks of a pendant tree very efficiently.

Lemma 2. Let $AB$ be an edge of a pendant tree $T$ and let $a_{\text{max}}$ and $b_{\text{max}}$ be vertices in $A$ and $B$ of maximum degrees, respectively. Then $\{a_{\text{max}}, b_{\text{max}}\}$ is non-pendant.

Proof. By Condition (ii) of Definition 1, there are vertices $a \in A$ and $b \in B$ such that $\lambda(a, b) < \min\{d(a), d(b)\}$. Since $\{a, a_{\text{max}}\}$ and $\{b, b_{\text{max}}\}$ are pendant, i.e.

$$\lambda(a, a_{\text{max}}) = \min\{d(a), d(a_{\text{max}})\} = d(a)$$

and $\lambda(b, b_{\text{max}}) = d(b)$, a minimum $a$-$b$-cut of size less than $\min\{d(a), d(b)\}$ can neither separate $a$ from $a_{\text{max}}$ nor $b$ from $b_{\text{max}}$. Hence,

$$\lambda(a_{\text{max}}, b_{\text{max}}) \leq \lambda(a, b) < \min\{d(a), d(b)\} \leq \min\{d(a_{\text{max}}), d(b_{\text{max}})\}.$$

Condition (iii) of pendant trees gives the following lemma.
Lemma 3. Let $AB$ be an edge of a pendant tree $T$ and let $a_{\text{max}}$ be a vertex in $A$ of maximum degree. Then $c(AB) < d(a_{\text{max}})$.

Proof. Let $b_{\text{max}}$ be a vertex of maximum degree in $B$ and let $a^* \in A$ and $b^* \in B$ be such that $c(AB) = \lambda(a^*, b^*)$ due to Condition (iii). By transitivity of the edge-connectivity $\lambda$, we have

$$\lambda(a_{\text{max}}, b_{\text{max}}) \geq \min\{\lambda(a_{\text{max}}, a^*), \lambda(a^*, b^*), \lambda(b^*, b_{\text{max}})\}$$

$$= \min\{d(a^*), \lambda(a^*, b^*), d(b^*)\}$$

$$= \lambda(a^*, b^*)$$

$$= c(AB),$$

where the first equality follows from the fact that $\{a_{\text{max}}, a^*\}$ and $\{b_{\text{max}}, b^*\}$ are pendant. According to Lemma 2, $\lambda(a_{\text{max}}, b_{\text{max}}) < d(a_{\text{max}})$, which gives the claim. 

We will construct a pendant tree by contracting edges in a Gomory-Hu tree. We recall that, given a graph $G$, a Gomory-Hu tree $T$ of $G$ is a tree on the vertex set $V(G)$, such that for every pair of vertices $a \neq b$ in $G$, there is an edge in the $a$-$b$-path in $T$ with that $E_G(V_T(C_e), V_T(C_f))$ is a minimum $a$-$b$-cut in $G$, where $C_e$ is a component obtained by deleting $e$ in $T$ and we denote by $V_T(C_e)$ the set of vertices in $G$ which are in the component $C_e$. In particular, $\lambda_G(a, b) = d_G(V_T(C_e))$. Here we see a Gomory-Hu tree not a tree on the vertex of $G$, but on the partition of $V(G)$ in which every part is a singleton.

Proposition 4. Given a Gomory-Hu tree of a graph $G$, a pendant tree of $G$ can be computed in linear time.

Proof. Let $T$ be a Gomory-Hu tree of $G$. Throughout the algorithm we maintain that every pair of distinct vertices in a block is pendant. We check iteratively for every edge $AB$ in $T$, whether there is a non-pendant pair $\{a, b\}$ with $a \in A$ and $b \in B$. We contract $AB$ in $T$ and set the new block as $A \cup B$ if and only if there is no such non-pendant pair. We claim that there is such a non-pendant pair if and only if $\min\{d_G(a_{\text{max}}), d_G(b_{\text{max}})\} > c(AB)$, where $a_{\text{max}}$ and $b_{\text{max}}$ are vertices in $A$ and $B$ with maximum degree, respectively. The sufficiency is clear (see also Lemma 2), and it suffices to show that if $\min\{d_G(a_{\text{max}}), d_G(b_{\text{max}})\} \leq c(AB)$, then $\{a, b\}$ is pendant for all $a \in A$ and $b \in B$.

Thus suppose $\min\{d_G(a_{\text{max}}), d_G(b_{\text{max}})\} \leq c(AB)$. Without loss of generality, let $d_G(a_{\text{max}}) \leq c(AB)$, which implies $d_G(a) \leq c(AB)$ for all $a \in A$. Let $a \in A$ and $b \in B$. By the property of Gomory-Hu trees, there are vertices $a^* \in A$ and $b^* \in B$ such that $\lambda_G(a^*, b^*) = c(AB)$; in particular, $d_G(b^*) \geq d_G(a^*) = c(AB)$. Then $\{a, b\}$ is pendant, since

$$\lambda_G(a, b) = \min\{\lambda_G(a, a^*), \lambda_G(a^*, b^*), \lambda_G(b^*, b)\}$$

$$= \min\{d_G(a), d_G(a^*), c(AB), d_G(b^*), d_G(b)\}$$

$$= \min\{d_G(a), d_G(b)\}.$$ 

The first equality comes from the transitivity of local edge-connectivity, the second comes from the fact that every vertex pair of a block is pendant, and the third holds, because $d_G(b^*) \geq d_G(a^*) = c(AB) \geq d_G(a)$.

It is not hard to see that the algorithm has a linear running time. 

In particular, Proposition 4 implies that every graph has a pendant tree as it is known that a Gomory-Hu tree always exists.
The best known running time for a deterministic construction of a Gomory-Hu tree is still based on the classical approach that applies \( n - 1 \) times the uncrossing technique to find uncrossing cuts on the input graph, and hence in \( O(n \theta_{\text{flow}}) \), where \( \theta_{\text{flow}} \) is the running time for a maximum flow subroutine (by Dinits’ algorithm [2, 5], \( \theta_{\text{flow}} = O(n^{2/3}m) \)).

Non-deterministically, Bhargat et al. [1] showed that a Gomory-Hu tree of a simple unweighted graph can be constructed in expected running time \( \tilde{O}(nm) \), where the tilde hides polylogarithmic factors.

Therefore, by our construction above, we conclude the following.

**Corollary 5.** Given a simple graph \( G \), a pendant tree of \( G \) can be computed deterministically in running time \( O(n^{5/3}m) \), and randomized in expected running time \( \tilde{O}(nm) \).

The next section gives several helpful lemmas that will be used in counting pendant pairs.

## 5 Large Blocks of Degree 1 and 2

For a tree \( T \) whose vertex set partitions \( V \), let \( V_k \) be the set of blocks of \( T \) having degree \( k \) in \( T \) and let \( V_{\geq k} := \bigcup_{k' \geq k} V_{k'} \). We call the blocks in \( V_1 \) leaf blocks. In \( T \), the set \( V_2 \) induces a family of disjoint paths; we call each such path a 2-path. We will prove that the leaf blocks of pendant trees as well as the blocks that are contained in 2-paths are large.

### Lemma 6. Let \( T \) be a pendant tree of a simple graph \( G \). Then every leaf block \( A \) of \( T \) satisfies \( |A| > \delta(G) \).

**Proof.** Let \( p := |A| \geq 1 \) and let \( B \) be the block adjacent to \( A \) in \( T \). By Lemma 3, we have \( \max_{v \in A} d(v) > c(AB) \geq \sum_{v \in A} (d(v) - (p - 1)) \geq \max_{v \in A} d(v) + \delta(p - 1) - p(p - 1) \), where the last inequality singles out the maximum degree. Therefore, \( p > 1 \) and \( p > \delta \).

Let \( a_{\max} \) be a vertex of maximal degree in a leaf block \( A \) with neighbor \( B \). Since \( c(AB) < d(a_{\max}) \), \( A \) must actually contain a vertex that has all its neighbors in \( A \), as otherwise each of the \( d(a_{\max}) \) incident edges of \( a_{\max} \) would contribute at least one edge to the edge-cut, either directly or by an incident edge of the corresponding neighbor of \( a_{\max} \). This gives the following corollary of Lemma 6, which was first shown by Mader.

### Corollary 7 ([9]). Let \( T \) be a pendant tree of a simple graph \( G \). Then every leaf block \( A \) contains a vertex \( v \) with \( N(v) \subseteq A \). Hence, every pair in \( \{v\} \cup N(v) \) is pendant.

This already implies that simple graphs contain \( (\delta + 1)^2 = \Omega(\delta^2) \) pendant pairs. Note that Lemma 6 and Corollary 7 do not hold for graphs having parallel edges: for example, consider a block \( A \) that consists of two vertices of degree \( \delta \), which are joined by \( \delta - 1 \) parallel edges. However, even if the graph is not simple, a leaf block \( A \) must always contain at least two vertices due to Lemma 3.

### Corollary 8. Every leaf block of a pendant tree of a graph contains at least two vertices.

In simple graphs, we thus know that leaf blocks give us a large number of pendant pairs. Since \( T \) is a tree, the number of leaf blocks is completely determined by the number of blocks of degree at least 3, namely \( |V_3| = \sum_{A \in V_{\geq 3}} (d_T(A) - 2) + 2 \). Thus, in order to prove a better lower bound on the number of pendant pairs, we have to consider the case that there are many small blocks of size \( o(\delta) \) contained in 2-paths. The following two lemmas prove that (i) for every two adjacent blocks \( A \) and \( B \) in a 2-path with \( |A| + |B| > 2 \), we have \( |A| + |B| \geq \delta - 1 = \Omega(\delta) \) and (ii) if \( \delta \geq 5 \) and \( P \) is a subpath of a 2-path such that all blocks
of $P$ are singletons, then $P$ contains at most two blocks. This will be used later to show that the bad situation of many small blocks of size $o(\delta)$ can actually not occur. We omit the proofs in this extended abstract.

**Lemma 9.** Let $T$ be a pendant tree of a simple graph $G$. Let $AB$ be an edge in $T$ with $A, B \in V_T$. If $|A| + |B| > 2$, $|A| + |B| \geq \delta(G) - 1$.

**Lemma 10.** Let $T$ be a pendant tree of a simple graph $G$ with $|V(T)| > 1$. Let $A = \{v_A\}$ be a block in $V_T$ with neighborhood $B_1, \ldots, B_r \in V_2$ in $T$ such that $|A| = |B_1| = \cdots = |B_r| = 1$. Let $B_i' \neq A$ be the block that is adjacent to $B_i$ in $T$. Then $d(v_A) \leq r^2 - 2\gamma$, where $\gamma := \sum_{1 \leq i < j \leq r} d(C_{B_i' B_j'}, C_{B_i' B_j'})$ is the number of cross-edges. In particular, we have $\delta(G) \leq r^2$ and $\lambda(G) < r^2$. Moreover, if $r = 2$, $\kappa(G) \leq 2$.

Setting $r = 2$ in Lemma 10 gives the following corollary for adjacent blocks of 2-paths. Note that the proof of Lemma 10 allows to weaken the conditions of this corollary further if the number of cross-edges is large.

**Corollary 11.** Let $G$ be simple and let $AB$ and $BC$ be edges in a 2-path of $T$. If $\delta(G) \geq 5$ or $\lambda(G) \geq 4$ or $\kappa(G) \geq 3$, then $|A| + |B| + |C| > 3$.

For every block $A \in V_2$, let $A$ be in $V_2^{in}$ if all of its neighbors are also in $V_2$; otherwise, let $A$ be in $V_2^{out}$. The blocks in $V_2^{out}$ are exactly the endblocks of 2-paths.

**Lemma 12.** Let $T$ be a tree. If $|V(T)| > 1$, then $|V_{>2}| \leq |V_1| - 2$ and $|V_2^{out}| \leq 4|V_1| - 6$.

Now we are ready to show that the blocks of 2-paths contain many vertices if $\delta(G) \geq 5$ or $\lambda(G) \geq 4$ or $\kappa(G) \geq 3$.

**Lemma 13.** Let $T$ be a pendant tree of a simple graph $G$ satisfying $\delta(G) \geq 5$ or $\lambda(G) \geq 4$ or $\kappa(G) \geq 3$. Let $P$ be a 2-path of $T$. Then

$$\sum_{S \in V(P)} |S| \geq (|V(P)| - 2) \frac{\max\{4, \delta(G)\}}{3} + 2.$$

We will use these lemmas to count pendant pairs in the next section.

## 6 Many Pendant Pairs

We will use the results on large blocks of the previous section to obtain our main theorems, Theorems 15 and 16. While the latter shows the existence of $\Omega(\delta n)$ pendant pairs, as mentioned in the introduction, the former gives the slightly weaker bound $\Omega(n)$, but in return counts only pendant pairs of a special type.

**Definition 14.** Let a set $F$ of pendant pairs be dependent if $V$ contains at least three distinct vertices $v_1, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in F$ for all $i = 1, \ldots, k$, where we set $v_{k+1} := v_1$; otherwise, $F$ is called independent.

Counting only independent pendant pairs allows us to deduce statements about the number of vertices in the graph that is obtained from contracting these pairs (these are not true for arbitrary sets of pendant pairs): Theorem 15 will prove for $\delta \geq 5$ that there are at least $\frac{1}{211} n \geq \frac{1}{211} n = \Omega(n)$ such independent pendant pairs. We will show that the contractions imply not only an additive decrease of the number of vertices by at least $\frac{1}{11} n$, but also a multiplicative decrease by the factor $\delta$ (i.e. the number of vertices left is $O(n/\delta)$). We omit the proof in this extended abstract.
Theorem 15. Let \( G \) be a simple graph that satisfies \( \delta(G) \geq 5 \) or \( \lambda(G) \geq 4 \) or \( \kappa(G) \geq 3 \). Let \( T \) be a pendant tree of \( G \). Then \( G \) has at least \( \frac{1}{30}n = \Omega(n) \) independent pendant pairs each of which is in some block of \( T \) and whose pairwise contraction leaves \( O(n/\delta) \) vertices in the graph.

For arbitrary pendant pairs not requiring independence, we improve the lower bound \( \Omega(n) \) of Theorem 15 to \( \Omega(\delta n) \) in the following theorem. This is done by grouping the blocks more precisely. The main idea is that the blocks are of average size \( \Omega(\delta) \) and therefore contain \( \Omega(\delta^2) \) pendant pairs on average. As the number of blocks is \( O(n/\delta) \), we thus expect that the number of pendant pairs is \( \Omega(\frac{2}{3} \cdot \delta^2) = \Omega(\delta n) \).

Theorem 16. Let \( G \) be a simple graph that satisfies \( \delta(G) \geq 5 \) or \( \lambda(G) \geq 4 \) or \( \kappa(G) \geq 3 \). Then \( G \) contains at least \( \frac{1}{30} \delta n = \Omega(\delta n) \) pendant pairs.

Proof. Note that \( n > \delta \geq 3 \). If \( G \) does not contain a non-pendant pair, there are \( \frac{\binom{n}{2}}{\delta^2} \) pendant pairs in \( G \). Otherwise, \( G \) contains a non-pendant pair. Let \( T \) be a pendant tree of \( G \); then \( |V(T)| \geq 2 \).

For each 2-path \( P \) with \( |V(P)| \geq 3 \), let \( P^* \) be a subpath obtained from \( P \) by deleting at most two endblocks (i.e. blocks in \( P \cap V_2^\text{out} \)) of \( P \) such that \( |V(P^*)| \) is a multiple of 3. Then, we split \( P^* \) into subpaths \( P_1^*, \ldots, P_{|V(P^*)|/3}^* \), each of size 3. Now, let \( M_P \) be a collection of blocks that contains exactly one block \( S_i \in V(P_i^*) \) for every \( i = 1, \ldots, \frac{|V(P^*)|}{3} \), such that \( S_i \) is of maximum size amongst other blocks in \( V(P_i^*) \). By Corollary 11 and Lemma 9, every block \( S \in M_P \) is of size at least \( \max\{2, (\delta - 1)/2\} \).

Let \( V_2^* := V_2 - \bigcup_{2\text{-path } P, |V(P)| \geq 3} V(P^*) \subseteq V_2^\text{out} \). For every leaf block \( S \in V_1 \), let \( Y_S \) be a collection of blocks that consists of \( S \), at most four blocks from \( V_2^* \) and at most one block from \( V_2 \) such that the collections \( Y_S \) (\( S \in V_1 \)) form a partition of \( V_1 \cup V_2^* \cup V_2 \); such allocation exists as \( |V_2^*| \leq |V_2^\text{out}| \leq 4|V_1| \) and \( |V_2| \leq |V_1| \) (Lemma 12). For every \( S \in V_1 \), let \( D_S \) be a block in \( Y_S \) of maximum size. Then, by Lemma 6, \( |D_S| \geq |S| > \delta \).

Now we can count the number of pendant pairs to obtain the desired lower bound, as the blocks have average size \( \Omega(\delta) \). The number of pendant pairs in \( G \) is at least

\[
\sum_{S \in V(T)} \left( \frac{|S|}{2} \right)
\geq \sum_{S \in V_1} \frac{|D_S|(|D_S| - 1)}{2} + \sum_{2\text{-path } P, |V(P)| \geq 3} \sum_{S \in M_P} \frac{|S|(|S| - 1)}{2}
\geq \frac{\delta}{2} \sum_{S \in V_1} |D_S| + \frac{\delta}{10} \sum_{2\text{-path } P, |V(P)| \geq 3} \sum_{S \in M_P} |S|
\geq \frac{\delta}{2} \sum_{S \in V_1 \cup V_2^* \cup V_2} |S| + \frac{\delta}{10} \sum_{S \in V_2^* \cup V_2} |S|
\geq \frac{1}{30} \delta n = \Omega(\delta n).
\]

We remark that the constants 1/12 and 1/30 in the proofs of the bounds of Theorems 15 and 16 can be improved for larger \( \delta \).
A Cut Tree Representation for Pendant Pairs

Figure 1 The bone graph $G$, whose only pendant pairs are the ones contained in the two $K_5$ (those form the only leaf blocks of the pendant pair tree). Hence, $G$ has exactly 20 pendant pairs.

7 Tightness

Clearly, any graph $G$ contains at most $n - 1$ independent pendant pairs, hence the order of the lower bound in Theorem 15 is best possible. The order of the number of vertices left after contraction in Theorem 15 and that of the number of pendant pairs in Theorem 16 are also tight; consider the unions of $\frac{1}{\sqrt{n}}$ many disjoint cliques $K_{n+1}$.

Each of the conditions $\delta \geq 5$, $\lambda \geq 4$ and $\kappa \geq 3$ in Theorems 15 and 16 is tight, as the graph in Figure 1 can be arbitrarily large and satisfies $\delta = 4$, $\lambda = 3$ and $\kappa = 2$ but has only a constant number of pendant pairs. Also the simpleness condition in both results is indispensable: Consider the path graph on $n$ vertices whose two end edges have multiplicity $\delta$ and all other edges have multiplicity $\delta/2$. This graph has precisely 2 pendant pairs, each at one of its ends.

References


Polyline Drawings with Topological Constraints

Emilio Di Giacomo  
Università degli Studi di Perugia, Perugia, Italy  
emilio.digiacomo@unipg.it  
https://orcid.org/0000-0002-9794-1928

Peter Eades  
University of Sydney, Sydney, Australia  
peter.d.eades@gmail.com

Giuseppe Liotta  
Università degli Studi di Perugia, Perugia, Italy  
giuseppe.liotta@unipg.it  
https://orcid.org/0000-0002-2886-9694

Henk Meijer  
University College Roosevelt, Middelburg, The Netherlands  
h.meijer@ucr.nl

Fabrizio Montecchiani  
Università degli Studi di Perugia, Perugia, Italy  
fabrizio.montecchiani@unipg.it  
https://orcid.org/0000-0002-0543-8912

Abstract

Let $G$ be a simple topological graph and let $\Gamma$ be a polyline drawing of $G$. We say that $\Gamma$ partially preserves the topology of $G$ if it has the same external boundary, the same rotation system, and the same set of crossings as $G$. Drawing $\Gamma$ fully preserves the topology of $G$ if the planarization of $G$ and the planarization of $\Gamma$ have the same planar embedding. We show that if the set of crossing-free edges of $G$ forms a connected spanning subgraph, then $G$ admits a polyline drawing that partially preserves its topology and that has curve complexity at most three (i.e., at most three bends per edge). If, however, the set of crossing-free edges of $G$ is not a connected spanning subgraph, the curve complexity may be $\Omega(\sqrt{n})$. Concerning drawings that fully preserve the topology, we show that if $G$ has skewness $k$, it admits one such drawing with curve complexity at most $2k$; for skewness-1 graphs, the curve complexity can be reduced to one, which is a tight bound. We also consider optimal 2-plane graphs and discuss trade-offs between curve complexity and crossing angle resolution of drawings that fully preserve the topology.

2012 ACM Subject Classification Mathematics of computing → Combinatorics, Mathematics of computing → Graph theory, Theory of computation → Design and analysis of algorithms

Keywords and phrases Topological graphs, graph drawing, curve complexity, skewness-$k$ graphs, $k$-planar graphs

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.39

Related Version A full version of the paper is available at [13], https://arxiv.org/abs/1809.08111.

Acknowledgements We wish to thank Stephen Wismath for useful discussions.
1 Introduction

A fundamental result in graph drawing is the so-called “stretchability theorem” [12, 17, 18]: Every planar simple topological graph admits a straight-line drawing that preserves its topology. One may ask whether a similar theorem holds for non-planar simple topological graphs. Motivated by the fact that a straight-line drawing may not be possible even for a planar graph plus an edge [10], we allow bends along the edges and measure the quality of the computed drawings in terms of their curve complexity, defined as the maximum number of bends per edge.

Let $G$ be a simple topological graph and let $\Gamma$ be a polyline drawing of $G$. (Note that, by definition of simple topological graph, $G$ has neither multiple edges nor self-loops; see also Section 2 for formal definitions.) Drawing $\Gamma$ fully preserves the topology of $G$ if the planarization of $G$ (i.e., the planar simple topological graph obtained from $G$ by replacing crossings with dummy vertices) and the planarization of $\Gamma$ have the same planar embedding. Eppstein et al. [11] prove the existence of a simple arrangement of $n$ pseudolines that, when drawn with polylines, it requires at least one pseudoline to have $\Omega(n)$ bends. It is not hard to see that the result by Eppstein et al. implies the existence of an $n$-vertex simple topological graph such that any polyline drawing that fully preserves its topology has curve complexity $\Omega(n)$ (see Corollary 2 in Section 2). This lower bound naturally suggests two research directions: (i) “Trade” curve complexity for accuracy in the preservation of the topology and (ii) Describe families of simple topological graphs for which polyline drawings that fully preserve their topologies and that have low curve complexity can be computed.

Concerning the first research direction, we consider the following relaxation of topology preserving drawing. A polyline drawing of a simple topological graph $G$ partially preserves the topology of $G$ if it has the same rotation system, the same external boundary, and the same set of crossings as $G$, while it may not preserve the order of the crossings along an edge. It may be worth recalling that some (weaker) notions of topological equivalence between graphs have been already considered in the literature. For example, Kynčl [15, 16] and Aichholzer et al. [1, 2] study weakly isomorphic simple topological graphs: Two simple topological graphs are weakly isomorphic if they have the same set of vertices, the same set of edges, and the same set of edge crossings. Note that a drawing $\Gamma$ that partially preserves the topology of a simple topological graph $G$ is weakly isomorphic to $G$ and, in addition, it has the same rotation system and the same external boundary as $G$. Also, Kratochvíl, Lubiw, and Nešetřil [14] define the notion of abstract topological graph as a pair $(G, \chi)$, where $G$ is a graph and $\chi$ is a set of pairs of crossing edges; a strong realization of $G$ is a drawing $\Gamma$ of $G$ such that two edges of $\Gamma$ cross if and only if they belong to $\chi$. The problem of computing a drawing that partially preserves a topology may be rephrased as the problem of computing a strong realization of an abstract topological graph for which a rotation system and an
external boundary are given in input. A different relaxation of the topology preservation is studied by Durocher and Mondal, who proved bounds on the curve complexity of drawings that preserve the thickness of the input graph [9].

Concerning the second research direction, we investigate the curve complexity of polyline drawings that fully preserve the topology of meaningful families of beyond-planar graphs, that are families of non-planar graphs for which some crossing configurations are forbidden (see, e.g., [4, 8] for surveys and special issues on beyond-planar graph drawing). In particular, we focus on graphs with skewness $k$, i.e., non-planar graphs that can be made planar by removing at most $k$ edges, and on 2-plane graphs, i.e., non-planar graphs for which any edge is crossed at most twice. Note that a characterization of those graphs with skewness one having a straight-line drawing that fully preserves the topology is presented in [10]. Also, all 1-plane graphs (every edge can be crossed at most once) admit a polyline drawing with curve complexity one that fully preserves the topology and such that any crossing angle is $\frac{\pi}{2}$ [6].

Our results can be listed as follows. Let $G$ be a simple topological graph.

- If the subgraph of $G$ formed by the uncrossed edges and all vertices of $G$, called planar skeleton, is connected, then $G$ admits a polyline drawing with curve complexity three that partially preserves its topology. If the planar skeleton is biconnected the curve complexity can be reduced to one, which is worst-case optimal (Section 3).

- For the case that the planar skeleton of $G$ is not connected, we prove that the curve complexity may be $\Omega(\sqrt{n})$ (Section 3).

- If $G$ has skewness $k$, then $G$ admits a polyline drawing with curve complexity $2k$ that fully preserves its topology. When $k = 1$, the curve complexity can be reduced to one, which is worst-case optimal (Section 4).

- If $G$ is optimal 2-plane (i.e., it is 2-plane and it has $5n - 10$ edges), then $G$ admits a drawing that fully preserves its topology and with two bends in total, and a drawing that fully preserves its topology, with at most two bends per edge, and with optimal crossing angle resolution. The number of bends per edge can be reduced to one while maintaining the crossing angles arbitrarily close to $\frac{\pi}{2}$ (Section 4).

We conclude the introduction with an example about the difference between a drawing that fully preserves and one that partially preserves a given topology. Figure 1a shows a simple topological graph for which every polyline drawing fully preserving its topology has at least one bend on some edge. Figure 1b shows a drawing of the same graph that partially preserve its topology and has no bends.

For space reasons some proofs have been omitted and the corresponding statements are marked with an asterisk (*). Missing details can be found in [13].

## 2 Preliminaries

A simple topological graph is a drawing of a graph in the plane such that: (i) vertices are distinct points, (ii) edges are Jordan arcs that connect their endvertices and do not pass through other vertices, (iii) any two edges intersect at most once by either making a proper crossing or by sharing a common endvertex, and (iv) no three edges pass through the same crossing. A simple topological graph has neither multiple edges (otherwise there would be two edges intersecting twice), nor self-loops (because the endpoints of a Jordan arc do not coincide). A simple topological graph is planar if no two of its edges cross. A planar simple topological graph $G$ partitions the plane into topological connected regions, called faces of $G$. The unbounded face is called the external face. The planar embedding of a simple planar topological graph $G$ fixes the rotation system of $G$, defined as the clockwise circular order of
Polyline Drawings with Topological Constraints

Figure 2 (a) An arrangement of pseudolines \( \mathcal{L} \). (b) The graph \( G_\mathcal{L} \) associated with \( \mathcal{L} \).

the edges around each vertex, and the external face of \( G \). The planar skeleton of a simple topological graph \( G \) is the subgraph of \( G \) that contains all vertices and only the uncrossed edges of \( G \). A simple topological graph obtained from \( G \) by adding uncrossed edges (possibly none) is called a planar augmentation of \( G \).

Let \( \mathcal{L} \) be an arrangement of \( n \) pseudolines; a polyline realization \( \Gamma_\mathcal{L} \) of \( \mathcal{L} \) represents each pseudoline as a polygonal chain while preserving the topology of \( \mathcal{L} \). The curve complexity of \( \Gamma_\mathcal{L} \) is the maximum number of bends per pseudoline in \( \Gamma_\mathcal{L} \). The curve complexity of \( \mathcal{L} \) is the minimum curve complexity over all polyline realizations of \( \mathcal{L} \). The graph associated with \( \mathcal{L} \) is a simple topological graph \( G_\mathcal{L} \) defined as follows. Let \( C \) be a circle of sufficiently large radius such that all crossings of \( \mathcal{L} \) are inside \( C \) and every pseudoline intersects the boundary of \( C \) exactly twice. Replace each crossing between \( C \) and a pseudoline with a vertex, remove the portions of each pseudoline that are outside \( C \), add an apex vertex \( v \) outside \( C \), and connect \( v \) to the vertices of \( C \) with crossing-free edges. See Fig. 2 for an example.

\[ \text{Lemma 1 (*).} \] Let \( \mathcal{L} \) be an arrangement of \( n \) pseudolines and let \( G_\mathcal{L} \) be the simple topological graph associated with \( \mathcal{L} \). Every polyline drawing of \( G_\mathcal{L} \) that fully preserves its topology has curve complexity \( \Omega(f(n)) \) if and only if \( \mathcal{L} \) has curve complexity \( \Omega(f(n)) \).

Lemma 1 and the result of Eppstein et al. [11] proving the existence of an arrangement of \( n \) pseudolines with curve complexity \( \Omega(n) \) imply the following.

\[ \text{Corollary 2.} \] There exists a simple topological graph with \( n \) vertices such that any drawing that fully preserves its topology has curve complexity \( \Omega(n) \).

In the next section we study a relaxation of the concept of topology preservation by which we derive constant upper bounds on the curve complexity.

### 3 Polyline Drawings that Partially Preserve the Topology

A polygon \( P \) is star-shaped if there exists a set of points, called the kernel of \( P \), such that for every point \( z \) in this set and for each point \( p \) of on the boundary of \( P \), the segment \( \overline{zp} \) lies entirely within \( P \). A simple topological graph is outer if all its vertices are on the external boundary and all the edges of the external boundary are uncrossed. Let \( G \) be an outer simple topological graph with \( n \geq 3 \) vertices and let \( P \) be a star-shaped \( n \)-gon. A drawing \( \Gamma \) of \( G \) that extends \( P \) is such that the \( n \) vertices of \( G \) are placed at the corners of \( P \), and every edge of \( G \) is drawn either as a side of \( P \) or inside \( P \).

\[ \text{Lemma 3.} \] Let \( G \) be an outer simple topological graph with \( n \geq 3 \) vertices and let \( P \) be a star-shaped \( n \)-gon. There exists a polyline drawing of \( G \) with curve complexity at most one that partially preserves the topology of \( G \) and that extends \( P \).
We place the bend point of edge \((v_i, v_j)\). (b) Case 1: \((v_h, v_i)\) is contained in \(P_2\). (c) Case 2: \((v_h, v_i)\) intersects \((v_i, v_j)\).

**Proof.** We explain how to compute a drawing with the desired properties for the complete graph \(K_n\). Clearly a drawing of \(G\) can be obtained by removing the missing edges. Identify each vertex of \(K_n\) with a distinct corner of \(P\), and let \(\{v_0, v_1, \ldots, v_{n-1}\}\) be the \(n\) vertices of \(K_n\) in the clockwise circular order they appear along the boundary of \(P\). Note that every edge \((v_i, v_{i+1})\), for \(i = 0, 1, \ldots, n - 1\) (indices taken modulo \(n\)), coincides with a side of \(P\) and hence it is drawn as a straight-line segment. We now show how to draw all the edges between vertices at distance greater than one. The distance between two vertices \(v_i\) and \(v_j\) is the number of vertices encountered along \(P\) when walking clockwise from \(v_i\) (excluded) to \(v_j\) (included). We orient each edge \((v_i, v_j)\) from \(v_i\) to \(v_j\) if the distance between \(v_i\) and \(v_j\) is smaller than or equal to the distance between \(v_i\) and \(v_j\) is equal to the distance between \(v_i\) and \(v_j\). We add all oriented edges \((v_i, v_j)\) by increasing value of the span. Let \(c\) be an interior point of the kernel, for example its centroid. For any pair of vertices \(v_i\) and \(v_j\), let \(h_{i,j}\) be the bisector of the angle swept by \(r_i = \overrightarrow{ci}\) when rotated clockwise around \(c\) until it overlaps with \(r_j = \overrightarrow{cj}\). We denote by \(\Gamma_k\) the drawing after the addition of the first \(k \geq 0\) edges and maintain the following invariant for \(\Gamma_k\):

- For each oriented edge \((v_i, v_j)\) not yet in \(\Gamma_k\), there is a point \(p_{i,j}\) on \(h_{i,j}\) such that \((v_i, v_j)\) can be drawn with a bend at any point of the segment \(\sigma_{i,j} = \overrightarrow{p_{i,j}}\) intersecting any edge of \(\Gamma_k\) at most once (either at a crossing or at a common endpoint).

We will refer to the segment \(\sigma_{i,j}\) described in the invariant as the *free segment* of \((v_i, v_j)\).

Since \(P\) is star-shaped, the invariant holds for \(\Gamma_0\); in particular the free segment of every \((v_i, v_j)\) is the intersection of \(h_{i,j}\) with the kernel.

Let \((v_i, v_j)\) be the \(k\)-th edge to be added and assume that the invariant holds for \(\Gamma_{k-1}\). We place the bend point of \((v_i, v_j)\) at any point of the segment \(\sigma_{i,j}\). By the invariant, the resulting edge intersects any other existing edge at most once. We now prove that the invariant is maintained. The drawing of the edge \((v_i, v_j)\) divides the polygon \(P\) in two sub-polygons (see Fig. 3a). We denote by \(P_1\) the one that contains the portion of the boundary of \(P\) that is traversed when going clockwise from \(v_i\) to \(v_j\), and by \(P_2\) the other one. Notice that the point \(c\) is contained in \(P_2\). Let \((v_h, v_i)\) be any oriented edge not in \(\Gamma_k\). Before the addition of \((v_i, v_j)\), by the invariant there was a free segment \(\sigma_{h,l}\) for \((v_h, v_i)\). By construction, \((v_i, v_j)\) intersects \(\sigma_{h,l}\) at most once. If \((v_i, v_j)\) and \(\sigma_{h,l}\) intersect in a point \(p\), let \(p'\) be any point between \(c\) and \(p\) on \(\sigma_{h,l}\) and let \(\sigma'_{h,l} = \overrightarrow{cp'}\); if they do not intersect let \(\sigma'_{h,l} = \sigma_{h,l}\). In both cases \(\sigma'_{h,l}\) is completely contained in \(P_2\). We claim that \(\sigma'_{h,l}\) is a free
Figure 4  A simple topological graph with a triconnected planar skeleton that does not admit a straight-line drawing that partially preserves its topology.

From the argument above we obtain that the final drawing of $K_n$ has curve complexity one and extends $P$. By removing the edges of $K_n$ not in $G$, we obtain a polyline drawing $\Gamma$ of $G$ with curve complexity one that extends $P$. Moreover, $\Gamma$ partially preserves the topology of $G$. Namely, the circular order of the edges around each vertex and the external boundary are preserved by construction. Furthermore, since $G$ is outer, any two of its edges cross if and only if their four end-vertices appear interleaved when walking along its external boundary. This property is preserved in $\Gamma$, because the order of the vertices along $P$ is the same as the order of the vertices along the external boundary of $G$, and because any two edges cross at most once (either at a crossing or at a common endpoint).

We use Lemma 3 to compute a polyline drawing $\Gamma$ with constant curve complexity for any simple topological graph $G$ that has a biconnected planar skeleton $\sigma(G)$. We triangulate each face of $\sigma(G)$ and compute a straight-line drawing of this triangulation, which contains a drawing of $\sigma(G)$ where each face is a star-shaped polygon. Then, since each edge of $G \setminus \sigma(G)$ is inside one face of $\sigma(G)$, we draw these edges by using Lemma 3. Drawing $\Gamma$ has curve complexity one, which is worst-case optimal, even if the planar skeleton is triconnected (see, e.g., Fig. 4).

Theorem 4 (*). Let $G$ be a simple topological graph that admits a planar augmentation whose planar skeleton is biconnected. Then $G$ has a polyline drawing with curve complexity at most one that partially preserves its topology. The curve complexity is worst-case optimal.

If $\sigma(G)$ is connected, we can draw $G$ with three bends per edge.

Theorem 5 (*). Let $G$ be a simple topological graph that admits a planar augmentation whose planar skeleton is connected. Then $G$ has a polyline drawing with curve complexity at most three that partially preserves its topology.
Proof sketch. Let $G'$ be a planar augmentation of $G$ whose planar skeleton $\sigma(G')$ is connected. The idea is to add a set $E^*$ of edges to make $\sigma(G')$ biconnected and then use Theorem 4. For each face $f$ (possibly including the external one) whose boundary contains at least one cutvertex we execute the following procedure. Walk clockwise along the boundary of $f$ and let $v_0, v_1, v_2, \ldots, v_k$ be the sequence of vertices in the order they are encountered during this walk, where the vertices that are encountered more than once (i.e., the cutvertices) appear in the sequence only when they are encountered for the first time. For each pair of consecutive vertices $v_{i-1}$ and $v_i$ (for $i = 1, 2, \ldots, k$) in the above sequence, if $v_{i-1}$ and $v_i$ are not adjacent in $\sigma(G')$, add to $E^*$ the edge $(v_{i-1}, v_i)$. See Fig. 5a and 5b for an example. If we add the edges of $E^*$ to $G'$ (embedded in the same way with respect to $\sigma(G')$), we obtain a new topological graph such that the edges of $E^*$ cross the edges of $G' \setminus \sigma(G)$ (see Fig. 5c). Replacing each of the crossings created by the addition of $E^*$ with dummy vertices, we obtain a new topological graph $G''$ whose planar skeleton is biconnected. By Theorem 4 $G''$ admits a drawing that partially preserves its topology and such that each edge has at most one bend. Replacing dummy vertices with bends, we obtain a drawing of $G'$ that partially preserves its topology. An edge $e$ is split in at most three “pieces” in $G''$. The two “pieces” that are incident to the original vertices are not crossed in $G''$ and therefore they belong to $\sigma(G'')$ and are drawn without bends. The third “piece” is not in $\sigma(G'')$ and is drawn with at most one bend. Thus, $e$ has at most three bends. 

Theorems 4 and 5 show that constant curve complexity is sufficient for drawings that partially preserve the topology of graphs whose planar skeleton is connected. It is worth remarking that a drawing that fully preserves the topology may require $\Omega(n)$ curve complexity even if the planar skeleton is connected. Namely, the planar skeleton of the graphs associated with arrangements of pseudolines is always biconnected and, by Corollary 2, there exists one such graph that has $\Omega(n)$ curve complexity.

One may wonder whether the constant curve complexity bound of Theorems 4 and 5 can be extended to the case on non-connected planar skeletons. This question is answered in the negative by the next theorem.

\textbf{Theorem 6 (*).} There exists a simple topological graph with $n$ vertices such that any drawing that partially preserves its topology has curve complexity $\Omega(\sqrt{n})$.

Proof sketch. Let $\mathcal{L}$ be an arrangement of pseudolines and let $G_{\mathcal{L}}$ be the graph associated with $\mathcal{L}$. By Lemma 1 any drawing that fully preserves the topology of $G_{\mathcal{L}}$ cannot have a
Figure 6 (a) Straight-line drawing of the graph $G_L$ of Fig. 2b. (b) The graph $\overline{G}_L$ for the arrangement of Fig. 2a.

better curve complexity than $L$. On the other hand if we only want to partially preserve the topology, $G_L$ can be realized without bends (see Fig. 6a for a straight-line drawing of the graph of Fig. 2b). We now describe how to construct a supergraph $\overline{G}_L$ of $G_L$ such that in any drawing of $\overline{G}_L$ that partially preserves its topology, the topology of the subgraph $G_L$ is fully preserved. Refer to Fig. 6b for an illustration concerning the graph of Fig. 2b.

The set $E^*$ of crossing edges of $G_L$ forms a set of cells inside the cycle $C$ of $G_L$ (these cells correspond to the internal faces of the planarization of $G_L$). For each of these cells, we add a vertex inside the cell and we connect two such vertices if the corresponding cells share a side. For those cells that have as a side an edge $e$ of $C$ we add an edge between the vertex added inside that cell and the two end-vertices of $e$. Let $\overline{G}_L$ be the resulting topological graph and let $\Gamma_L$ be a drawing that partially preserves the topology of $\overline{G}_L$. It can be proved that the sub-drawing $\Gamma_L$ of $\overline{G}_L$ representing $G_L$ fully preserves the topology of $G_L$.

Denote by $L_N$ the arrangement of $N$ pseudolines defined by Eppstein et al. [11]. By the argument above, any polyline drawing that partially preserves the topology of the graph $\overline{G}_L$ contains a sub-drawing of $G_L$ that fully preserves its topology and hence has curve complexity $\Omega(N)$ by Lemma 1. The number of vertices of $G_{L_N}$ is $2N + 1$ and the number of cells is $\Theta(N^2)$. This implies that the number of vertices of $\overline{G}_{L_N}$ has $n = \Theta(N^2)$. Thus, any drawing that partially preserves the topology of $\overline{G}_{L_N}$ has curve complexity $\Omega(N) = \Omega(\sqrt{n})$.

Based on Theorem 6 one may wonder whether $O(\sqrt{n})$ curve complexity is sufficient when the skeleton is not connected. The following theorem states a preliminary result in this direction, extending Theorem 5 to the case that the planar skeleton consists of at most $c$ connected components.

**Theorem 7 (**). Let $G$ be a simple topological graph that admits a planar augmentation whose planar skeleton has $c$ connected components. Then $G$ has a polyline drawing with curve complexity at most $4c - 1$ that partially preserves its topology.

## 4 Polyline Drawings that Fully Preserve the Topology

In this section we study polyline drawings of constant curve complexity for two meaningful families of beyond-planar graphs. Namely, we consider $k$-skew graphs and 2-plane graphs. A simple topological graph $G = (V, E)$ is $k$-skew if there is a set $F \subseteq E$ of $k$ edges such that $G' = (V, E \setminus F)$ does not contain crossings. A simple topological graph is 2-plane if every edge is crossed by at most two other edges. A 2-plane graph with $n$ vertices can have at most $5n - 10$ edges and it is called *optimal 2-plane* if it has exactly $5n - 10$ edges. We prove that
the graphs belonging to these two families admit a polyline drawing that fully preserves the topology and has constant curve complexity. A tool that we are going to use is the algorithm of Chiba et al. [7] that receives as input a 3-connected plane graph $G$ whose external face has $k \geq 3$ vertices, and a convex polygon $P$ with $k$ corners. The algorithm computes a straight-line drawing $\Gamma$ of $G$ that fully preserves the topology of $G$, it has polygon $P$ as its external face, and all internal faces are convex. Moreover, if three consecutive vertices belong to a same face and are collinear in the computed drawing, we can slightly perturb one of them without destroying the convexity of the other faces. Thus, we can assume that all faces of $\Gamma$ are strictly convex.

We first show that a $k$-skew topological graph admits a polyline drawing that fully preserves the topology of $G$ and has at most $2k$ bends per edge. The technique is based on an approach that we call the sleeve method and that is illustrated in the following.

**The sleeve method.** Suppose that $G$ is a topological graph such that the removal of the edge $(s, t)$ makes $G$ without crossings. Let $E_\chi$ be the set of edges that cross $(s, t)$ and suppose that $\alpha$ is a crossing between edges $(s, t)$ and $(u, v) \in E_\chi$ in $G$. If the clockwise order of the vertices around $\alpha$ is $(s, u, t, v)$, then $u$ is a left vertex and $v$ is a right vertex (with respect to the ordered pair $(s, t)$ and the crossing $\alpha$). We add a “sleeve” around $(s, t)$, as follows. Number the edges of $E_\chi = \{e_1, e_2, \ldots, e_p\}$ in the order of their crossings $\alpha_1, \alpha_2, \ldots, \alpha_p$ along $(s, t)$, so that $e_i = (u_i, v_i)$ crosses $(s, t)$ at $\alpha_i$, $u_i$ is left, and $v_i$ is right. We subdivide each edge $(u_i, v_i)$ with dummy vertices $u_i'$ and $v_i'$ so that the edge $(u_i, v_i)$ becomes a path $(u_i, u_i', v_i', v_i)$ with the crossing point $\alpha_i$ in between $u_i'$ and $v_i'$. Note that after this subdivision, $u_i'$ is left and $v_i'$ is right, and $u_i$ and $v_i$ are neither left nor right. Next we add a path $p_L$ that begins at $s$ and visits each of the left dummy vertices $u_i'$ in the order $u_1, u_2, \ldots, u_p$, and ends at $t$. Similarly we add a path $p_R$ that visits $s$, all the right vertices, and then $t$. We call the cycle formed by $p_L$ and $p_R$ a sleeve. Note that the interior of the sleeve contains the edges $(u_i', v_i')$ and the edge $(s, t)$, but no other vertices or edges. The next theorem explains how to draw $k$-skew graphs with curve complexity $2k$.

**Theorem 8.** Every $k$-skew simple topological graph admits a polyline drawing with curve complexity at most $2k$ that fully preserves its topology.

**Proof.** Suppose that $G = (V, E)$ is a topological graph and there is a set $F \subseteq E$ of $k$ edges such that deleting all the edges in $F$ from $G$ gives a planar topological graph. An example with $k = 2$ is in Fig. 7a. Replace each crossing between a pair of edges in $F$ with a dummy vertex, and let $G'$ be the resulting graph. In $G'$ there is a set $F'$ of edges such that no two
Figure 8 (a) A 1-skew graph with an inconsistent vertex (larger and purple). (b) A 1-skew graph with an internal inconsistent face (shaded), in which every vertex is consistent.

edges in \( F' \) cross, and deleting all the edges in \( F' \) from \( G' \) gives a planar topological graph. Here \( |F'| \leq k + 2c \), where \( c \) is the number of crossings between edges in \( F \). Also, note that the number of such crossings on each edge in \( F \) is at most \( k - 1 \). Now add a sleeve around each edge \((s, t) \in F'\) using the sleeve method, and let \( G'' \) be the resulting graph (see Fig. 7b). Note that two such sleeves do not share any edge, and they share at most one vertex. Delete the interior of each sleeve in \( G'' \) to give a planar topological graph \( G''' \). Note that each sleeve of \( G'' \) gives a face of \( G''' \). Now triangulate \( G''' \) except for the faces of \( G''' \) formed by the sleeves (see Fig. 7c).

The resulting graph \( G''' \) is triconnected by Barnette’s Theorem [3], since two faces share at most one edge or at most one vertex. We can construct a planar drawing \( \Gamma''' \) of \( G''' \) using the convex drawing algorithm of Chiba et al. [7]. Each face of \( \Gamma''' \) is convex, including each face that comes from a sleeve. Drawing the edges of \( G'' \) inside each sleeve as straight-line segments gives a straight-line drawing of \( G'' \). Deleting the dummy edges of the sleeves, and replacing the dummy vertices of the sleeves by bends, we have a polyline drawing \( \Gamma' \) of \( G \) that fully preserves the embedding of \( G \). The only bends are (1) at the crossing points between edges of \( F \), and (2) at the dummy vertices of the sleeves. Let \( e \) be an edge of \( G \). If \( e \in E' \setminus F \), then \( e \) crosses at most \( k \) edges (those in \( F \)) and each of these crossings creates two dummy vertices in a sleeve of \( G'' \), thus resulting in \( 2k \) bends. If \( e \in F \), then it has bends at the crossings with other edges of \( F \), which are at most \( k - 1 \).

By Theorem 8 we can draw a 1-skew topological graph with two bends per edge. We now prove that these graphs can be drawn using only one bend per edge. To this aim we first recall some results from [10]. We say that a vertex is inconsistent with respect to the edge \((s, t) \) if it is both left and right with respect to \((s, t) \), and consistent otherwise. For example, the graph in Fig. 8a has an inconsistent vertex. Observe that in a straight-line drawing of a topological graph, an inconsistent vertex would have to be both left and right of the straight line through \( s \) and \( t \). This gives the following necessary condition.

Lemma 9 ([10]). A 1-skew simple topological graph with an inconsistent vertex has no straight-line drawing that fully preserves its topology.

Without additional assumptions, the converse of Lemma 9 is false. For an example, consider Fig. 8b; this graph has no straight-line drawing, even though all vertices are consistent. The problem is that the internal face \((s, u, t, v) \) has both left and right vertices; as such, this face is inconsistent. To explore the converse of Lemma 9, we can assume that the topological graph is maximal 1-skew (that is, no edge can be added while retaining the property of being 1-skew). Namely, it has been proven that every 1-skew simple topological graph \( G \) with no inconsistent vertices can be augmented with dummy edges so that the resulting graph has no inconsistent vertices, it is maximal 1-skew, and it fully preserves the
Theorem 11 (*). 

Every 1-skew simple topological graph admits a polyline drawing with curve complexity at most one that fully preserves its topology. The curve complexity is worst-case optimal.

Proof sketch. Instead of placing a sleeve around the edge \((s, t)\), we use a “half-sleeve”, as follows. Again let \(E_x\) be the set of edges that cross \((s, t)\). We 1-subdivide each edge \((u, v) \in E_x\) with a dummy vertex on the left side of the crossing between \((u, v)\) and \((s, t)\), then add a path \(p_L\) that begins at \(s\) and visits each of the left dummy vertices in the order that their incident edges cross \((s, t)\), and ends at \(t\). Denote the graph obtained from \(G\) by adding this “left half-sleeve” as above by \(G_{s,t}^L\). Similarly, we could add a “right half-sleeve” to obtain a topological graph \(G_{s,t}^R\). It is clear that every vertex in both \(G_{s,t}^L\) and \(G_{s,t}^R\) is consistent. Note also that we have only added one dummy vertex on each edge \((u, v) \in E_x\); we aim to draw each of these edges with only one bend per edge. However, it is not clear that the internal faces of \(G_{s,t}^L\) and \(G_{s,t}^R\) are consistent. Consider, for example, the graph \(G\) in Fig. 8(b). For this graph, Fig. 9 shows \(G_{s,t}^L\), \(G_{s,t}^R\), \(G_{s,t}^L_{LR}\) and \(G_{s,t}^R_{LR}\). Note that \(G_{s,t}^L_{LR}\) has an internal inconsistent face, while \(G_{s,t}^R_{LR}\) does not. One can show that at most one of the graphs \(G_{s,t}^L_{LR}\) and \(G_{s,t}^R_{LR}\) has an internal inconsistent face. Thus, by Lemma 10, one of these two graphs admits a straight-line drawing which becomes a drawing with curve complexity one after the removal of the dummy vertices used to construct the half-sleeve.

We conclude this section with our results about optimal 2-plane graphs.
Theorem 12 (\textit{\textdagger}). Every optimal 2-plane graph has a polyline drawing $\Gamma$ that fully preserves its topology and that has one of the following properties:

(a) $\Gamma$ has two bends in total.
(b) $\Gamma$ has curve complexity one and every crossing angle is at least $\frac{\pi}{2} - \epsilon$, for any $\epsilon > 0$.
(c) $\Gamma$ has curve complexity two and every crossing angle is exactly $\frac{\pi}{2}$.

5 Open Problems

Theorem 6 proves a lower bound of $\Omega(\sqrt{n})$ on the curve complexity of polyline drawings that partially preserve the topology and that do not have a connected skeleton. It may be worth understanding whether this bound is tight.

Theorem 12 proves that for optimal 2-plane graphs a crossing angle resolution arbitrarily close to $\frac{\pi}{2}$ can be achieved with curve complexity one, while optimal crossing angle of $\frac{\pi}{2}$ is achieved at the expenses of curve complexity two. Can optimal crossing angle resolution and curve complexity one be simultaneously achieved? A positive answer to this question is known if the planar skeleton of the graph is a dodecahedron \cite{5}.

Finally, a natural research direction suggested by the research in this paper is to extend the study of the curve complexity of drawings that fully preserve the topology to other families of beyond-planar topological graphs. For example, it would be interesting to understand whether Theorem 12 can be extended to non-optimal 2-plane graphs.

References


Almost Optimal Algorithms for Diameter-Optimally Augmenting Trees

Davide Bilò
Department of Humanities and Social Sciences, University of Sassari
Via Roma 151, 07100 Sassari (SS), Italy
davide.bilo@uniss.it
https://orcid.org/0000-0003-3169-4300

Abstract

We consider the problem of augmenting an n-vertex tree with one shortcut in order to minimize the diameter of the resulting graph. The tree is embedded in an unknown space and we have access to an oracle that, when queried on a pair of vertices \( u \) and \( v \), reports the weight of the shortcut \( (u,v) \) in constant time. Previously, the problem was solved in \( O(n^2 \log^3 n) \) time for general weights [Oh and Ahn, ISAAC 2016], in \( O(n^2 \log n) \) time for trees embedded in a metric space [Große et al., arXiv:1607.05547], and in \( O(n \log n) \) time for paths embedded in a metric space [Wang, WADS 2017]. Furthermore, a \((1 + \epsilon)\)-approximation algorithm running in \( O(n + 1/\epsilon^3) \) has been designed for paths embedded in \( \mathbb{R}^d \), for constant values of \( d \) [Große et al., ICALP 2015].

The contribution of this paper is twofold: we address the problem for trees (not only paths) and we also improve upon all known results. More precisely, we design a time-optimal \( O(n^2) \) time algorithm for general weights. Moreover, for trees embedded in a metric space, we design (i) an exact \( O(n \log n) \) time algorithm and (ii) a \((1 + \epsilon)\)-approximation algorithm that runs in \( O(n + \epsilon^{-1} \log \epsilon^{-1}) \) time.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis, Theory of computation → Approximation algorithms analysis

Keywords and phrases Graph diameter, augmentation problem, trees, time-efficient algorithms

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.40

Related Version A full version of this paper can be found at https://arxiv.org/abs/1809.08822.

1 Introduction

Consider a tree \( T = (V(T), E(T)) \) of \( n \) vertices, with a weight \( \delta(u, v) > 0 \) associated with each edge \( (u, v) \in E(T) \), and let \( c : V(T)^2 \to \mathbb{R}_{\geq 0} \) be an unknown function that assigns a weight to each possible shortcut \( (u, v) \) we could add to \( T \). For a given path \( P \) of an edge-weighted graph \( G \), the length of \( P \) is given by the overall sum of its edge weights. We denote by \( d_G(u, v) \) the distance between \( u \) and \( v \) in \( G \), i.e., the length of a shortest path between \( u \) and \( v \) in \( G \). The diameter of \( G \) is the maximum distance between any two vertices in \( G \), that is \( \max_{u, v \in V(G)} d_G(u, v) \).

In this paper we consider the Diameter-Optimal Augmentation Problem (DOAP for short). More precisely, we are given an edge-weighted tree \( T \) and we want to find a shortcut \( (u, v) \) whose addition to \( T \) minimizes the diameter of the resulting (multi)graph, that we denote

---

1 If \( u \) and \( v \) are in two different connected components of \( G \), then \( d_G(u, v) = \infty \).
by $T + (u,v)$. We assume to have (unlimited access to) an oracle that is able to report the weight of a queried shortcut in $O(1)$ time.

DOAP has already been studied before and the best known results are the following:
- an $O(n^2 \log^3 n)$ time and $O(n)$ space algorithm and a lower bound of $\Omega(n^2)$ on the time complexity of any exact algorithm [16];
- an $O(n^2 \log n)$ time algorithm for trees embedded in a metric space [11];
- an $O(n \log n)$ time algorithm for paths embedded in a metric space [18];
- a $(1 + \epsilon)$-approximation algorithm that solves the problem in $O(n + 1/\epsilon^3)$ for paths embedded in the Euclidean (constant) $k$-dimensional space [10].

In this paper we improve upon (almost) all these results. More precisely:
- we design an $O(n^2)$ time and space algorithm that solves DOAP. We observe that the time complexity of our algorithm is optimal;
- we develop an $O(n \log n)$ time and $O(n)$ space algorithm that solves DOAP for trees embedded in a metric space;
- we provide a $(1 + \epsilon)$-approximation algorithm, running in $O(n + 1/\epsilon^2)$ time and using $O(n + 1/\epsilon)$ space, that solves DOAP for trees embedded in a metric space.

Our approaches are similar in spirit to the ones already used in [10, 11, 18], but we need many new key observations and novel algorithmic techniques to extend the results to trees. Our results leave open the problem of solving DOAP in $O(n^2)$ time and truly subquadratic space for general instances, and in $o(n \log n)$ time for trees embedded in a metric space.

**Other related work.** The variant of DOAP in which we want to minimize the continuous diameter, i.e., the diameter measured with respect to all the points of a tree (not only its vertices), has been also addressed. Oh and Ahn [16] designed an $O(n^2 \log^3 n)$ time and $O(n)$ space algorithm. De Carufel et al. [3] designed an $O(n)$ time algorithm for paths embedded in the Euclidean plane. Subsequently, De Carufel et al. [4] extended the results to trees embedded in the Euclidean plane by designing an $O(n \log n)$ time algorithm.

Several generalizations of DOAP in which the graph (not necessarily a tree) can be augmented with the addition of $k$ edges have also been studied. In the more general setting, the problem is NP-hard [17], not approximable within logarithmic factors [2], and some of its variants - parameterized w.r.t. the overall cost of added shortcuts and resulting diameter - are even W[2]-hard [8, 9]. Therefore, several approximation algorithms have been developed for all these variations [2, 5, 7, 8, 14]. Finally, upper and lower bounds on the values of the diameters of the augmented graphs have also been investigated in [1, 6, 13].

**Our approaches.** Große et al. [10] were the first to attack DOAP for paths embedded in a metric space. They gave an $O(n \log n)$ time algorithm for the corresponding search version of the problem:

For a given value $\lambda > 0$, either compute a shortcut whose addition to the path induces a graph of diameter at most $\lambda$, or return ⊥ if such a shortcut does not exist.

Then, by implementing their algorithm also in a parallel fashion and applying Megiddo’s parametric-search paradigm [15], they solved DOAP for paths embedded in a metric space in $O(n \log^3 n)$ time. Lately, Wang [18] improved upon this result in two ways. First, he solved the search version of the problem in linear time. Second, he developed an ad-hoc

---

2 More precisely, $c$ is a metric function and $\delta(u,v) = c(u,v)$, for every $(u,v) \in E(G)$.
algorithm that, using the algorithm for the search version of the problem black-box together with sorted-matrix searching techniques and range-minima data structure, is able to: (i) reduce the size of the solution-search-space from \( \binom{n}{2} \) to \( n \) in \( O(n \log n) \) time and linear space by first reducing the size of the solution-search-space from \( \binom{n}{2} \) to \( n \) and then by evaluating the quality of the leftover solutions in \( O(n \log n) \) time. However, differently from Wang’s approach, we use Hershberger data structure for computing the upper envelope of a set of linear functions [12] rather than a range-minima data structure. Furthermore, there are several issues we have to deal with due to the much more complex topology of trees. We solve some of these issues using a lemma proved in [11] about the existence of an optimal shortcut whose endvertices both belong to a diametral path of the tree. This allows us to reduce our DOAP instance to a node-weighted path instance of a similar problem, that we call WDOAP, in which the distance between two vertices is measured by adding the weights of the two considered vertices to the length of a shortest path between them, and the diameter is defined accordingly. However, it is not possible to use the algorithms presented in [10, 18] black-box to solve WDOAP. Therefore we need to design an ad-hoc algorithm whose correctness strongly relies on the structural properties of diametral paths and properties satisfied by node weights. Furthermore, most of the easy observations that can be done for paths become non-trivial lemmas that need formal proofs for trees.

Our time-optimal algorithm that solves DOAP for instances with general weights is based on the following important observations. We reduce, in \( O(n^2) \) time, a DOAP instance to another DOAP instance in which the function \( c \) is graph-metric, i.e., \( c \) is an almost metric function that satisfies a weaker version of the triangle inequality. Since our \( O(n \log n) \) time algorithm for DOAP instances embedded in a metric space also works for graph-metric spaces, we can use this algorithm black-box to solve the reduced DOAP instance in \( O(n \log n) \) time, thus solving the original DOAP instance in \( O(n^2) \) time.

Finally, the \((1 + \varepsilon)\)-approximation algorithm for trees embedded in a metric space is obtained by proving that the diameter of the tree is at most three times the diameter, say \( D^* \), of an optimal solution. This allows us to partition the vertices along a diametral path into \( O(1/\varepsilon) \) sets such that the distance between any two vertices of the same set is at most \( O(\varepsilon D^*) \). We choose a suitable representative vertex for each of the \( O(1/\varepsilon) \) sets and use our \( O(n \log n) \) time algorithm to find an optimal shortcut in the corresponding WDOAP instance restricted to the set of representative vertices. Since the representative vertices are \( O(1/\varepsilon) \), the optimal shortcut in the restricted WDOAP instance can be found in \( O(\varepsilon^{-1} \log \varepsilon^{-1}) \) time. Furthermore, because of the choice of the representative vertex, we can show that the shortcut returned is a \((1 + \varepsilon)\)-approximate solution for the (unrestricted) WDOAP instance of our problem, i.e., a \((1 + \varepsilon)\)-approximate solution for our original DOAP instance.

Due to the lack of space, in this paper we only describe the \( O(n \log n) \) time algorithm for the instances embedded in a metric space and we refer to https://arxiv.org/abs/1809.08822 for the full version of the paper. The rest of the paper is organized as follows: in Section 2 we present some preliminary results among which the reduction from general instances to graph-metric instances; in Section 3 we describe the reduction from DOAP to WDOAP together with further simplifications; in Section 4 we design an algorithm that solves a search version of WDOAP in linear time; in Section 5 we develop an algorithm that solves DOAP for trees embedded in a graph-metric space.
Diameter-Optimally Augmentation Problem

We observe that a metric cost function is also graph-metric, but the opposite does not hold in general (see Figure 1). The graph-metric closure induced by $c$ is a function $ar{c}$ such that, for every two vertices $u$ and $v$ of $G$, $\bar{c}(u, v) = \min \{d_G(u, u') + c(u', v') + d_G(v', v) \mid u', v' \in V(G)\}$. The following lemma shows that we can restrict DOAP to input instances where $c$ is graph-metric. We observe that the reduction holds for any graph and not only for trees.

Lemma 1. Solving the instance $\langle G, \delta, c \rangle$ of DOAP is equivalent to solving the instance $\langle G, \delta, \bar{c} \rangle$ of DOAP, where $\bar{c}$ is the graph-metric closure induced by $c$.

Next lemma shows the existence of an optimal shortcut whose endvertices are both on a diametral path of $T$ for the case in which $c$ is a graph-metric.

Lemma 2. Let $(T, \delta, c)$ be an instance of DOAP, where $c$ is a graph-metric, and let $P = (v_1, \ldots, v_N)$ be a diametral path of $T$. There always exists an optimal shortcut $(u^*, v^*)$ such that $u^*, v^* \in V(P)$.

Reduction from trees to node-weighted paths

In this section we show that a DOAP instance embedded in a graph-metric space can be reduced in linear time to a node-weighted instance of a similar problem. The Node-Weighted-Diameter-Optimal Augmentation Problem (WDOAP for short) is defined as follows:

Input: A path $P = (v_1, \ldots, v_N)$, with a weight $\delta(v_i, v_{i+1}) > 0$ associated with each edge $(v_i, v_{i+1})$ of $P$, a weight $w(v_i)$ associated with each vertex $v_i$ such that $0 \leq w(v_i) \leq \min\{d(v_i, v_1), d(v_i, v_N)\}$, and an oracle that is able to report the weight $c(v_i, v_j)$ of a queried shortcut in $O(1)$ time, where $c$ is a graph-metric;

Output: Two indices $i^*$ and $j^*$, with $1 \leq i^* < j^* \leq N$, that minimize the function

$$D(i, j) := \max_{1 \leq k < h \leq N} \{w(v_k) + d_{v_i, v_j}(v_k, v_h) + w(v_h)\}.$$ 

We observe that $w(v_1) = w(v_N) = 0$. Let $\langle T, \delta, c \rangle$ be a DOAP instance, where $c$ is a graph-metric. Let $P = (v_1, \ldots, v_N)$ be a diametral path of $T$, $T_i$ the tree containing $v_i$ in the forest obtained by removing the edges of $P$ from $T$, and $w(v_i) := \max_{v \in V(T_i)} d(v, v)$. We say that $\langle P, \delta, w, c \rangle$ is the WDOAP instance induced by $\langle T, \delta, c \rangle$ and $P$. The following lemma holds.
Lemma 3. The WDOAP instance $⟨P, δ, w, c⟩$ induced by $⟨T, δ, c⟩$ and $P$ can be computed in $O(n)$ time. Moreover, $\text{diam}(T + (v_i, v_j)) = D(i, j)$, for every $1 \leq i < j \leq N$.

3.1 Further simplifications

In the rest of the paper, we show how to solve WDOAP in $O(N \log N)$ time and linear space. To avoid heavy notation, from now on we denote a vertex $v_i$ by using its associated index $i$. All the lemmas contained in this subsection are non-trivial generalizations of observations made in [10] for paths. We start proving a useful lemma.

Lemma 4. Let $i, j$ be two indices such that $1 \leq i < j \leq N$. Let $I = \{1\} \cup \{k \mid i < k \leq N\}$ and let $J = \{N\} \cup \{h \mid 1 \leq h < j\}$. We have that

$$D(i, j) = \max_{k \in I, h \in J, k < h} \left\{w(k) + d_{i, j}(k, h) + w(h)\right\}.$$

As we will see in a short, Lemma 4 allows us to decompose the function $D(i, j)$ into four monotone parts. First of all, for every $i = 1, \ldots, N$, we define

$$\omega(i) := \max \{w(j) - d(i, j) \mid 1 \leq j \leq N\}.$$

Observe that, for every $1 \leq i < j \leq N$,

$$\omega(i) \leq \omega(j) + d(i, j). \quad \text{(node-triangle inequality)}$$

Furthermore, $\omega(i) \geq w(i)$, for every $1 \leq i \leq N$, which implies $\omega(1) = \omega(N) = 0$. The following lemma establishes the time complexity needed to compute all the values $\omega(i)$.

Lemma 5. All the values $\omega(i)$, with $1 \leq i \leq N$, can be computed in $O(N)$ time.

For the rest of this section, unless stated otherwise, $i$ and $j$ are such that $1 \leq i < j \leq N$. The four functions used to decompose $D(i, j)$ are the following (see also Figure 2)

- $U(i, j) := d(1, i) + c(i, j) + d(j, N)$
- $S(i, j) := \max_{i < h < j} \left(\omega(h) + \min \{d(1, h), d(1, i) + c(i, j) + d(h, j)\}\right)$
- $E(i, j) := \max_{i < k \leq j} \left(\omega(k) + \min \{d(k, N), d(j, N) + c(i, j) + d(i, k)\}\right)$
- $C(i, j) := \max_{i < k < h < j} \left(\omega(k) + \min \{d(k, h), d(i, k) + c(i, j) + d(h, j)\}\right) + \omega(h)$

Using both the graph-triangle inequality and the node-triangle inequality, we can observe that all the four functions are monotonic. More precisely:

- $U(i, j + 1) \leq U(i, j) \leq U(i + 1, j)$
- $S(i - 1, j) \leq S(i, j) \leq S(i, j + 1)$
- $E(i, j + 1) \leq E(i, j) \leq E(i - 1, j)$
- $C(i + 1, j) \leq C(i, j) \leq C(i, j + 1)$

Moreover, we can prove the following lemma.

Lemma 6. $D(i, j) = \max \{U(i, j), S(i, j), E(i, j), C(i, j)\}$.

The following lemma allows us to efficiently compute the values $U(i, j), S(i, j), E(i, j), C(i, j)$.

Lemma 7. After a $O(N)$-time precomputation phase, for every $1 \leq i < j \leq N$, $U(i, j)$ can be computed in $O(1)$ time, while both $S(i, j)$ and $E(i, j)$ can be computed in $O(\log N)$ time.
In this section we design an $O(N)$ time algorithm for the following search version of WDOAP:

For a given WDOAP instance $\langle P, \delta, \omega, c \rangle$, where $c$ is a graph-metric and $\omega$ satisfies the node-triangle inequality, and a real value $\lambda > 0$, either find two indices $1 \leq i < j \leq N$ such that $D(i,j) \leq \lambda$, or return $\bot$ if such two indices do not exist.

In the following we assume that $d(1,N) > \lambda$, as otherwise $D(i,j) \leq \lambda$ for any two indices $i$ and $j$. For the rest of this section, unless stated otherwise, $i$ and $j$ are two fixed indices such that $1 \leq i < j \leq N$. Let $1 < \mu_i \leq N$ be the minimum index, or $N+1$ if such an index does not exists, such that $U(i,\mu_i) \leq \lambda$. Our algorithm computes the index $\mu_i$, for every $1 \leq i < N$. As $U(i,j) \geq U(i,j+1)$ for every $i < j < N$, the following lemma holds.

Lemma 8. $U(i,j) \leq \lambda$ iff $\mu_i \leq j$ (see also Figure 3).

Moreover, as $U(i,j) \leq U(i+1,j)$, we have that $\mu_i \leq \mu_{i+1}$. Therefore, our algorithm can compute all the indices $\mu_i$ in $O(N)$ time by scanning all the vertices of $P$ from 1 to $N$.

We introduce some new notation useful to describe our algorithm. We define $r_i$ as the maximum index such that $i < r_i \leq N$ and $\omega(i) + d(i,r_i) + \omega(r_i) \leq \lambda$. If such an index does not exist, we set $r_i = i$. Similarly, we define $\ell_N$ as the minimum index such that $1 \leq \ell_N < N$ and $\omega(\ell_N) + d(\ell_N,N) \leq \lambda$. If such an index does not exist, we set $\ell_N = N$. Observe that if $j \leq r_i$, then, using the node-triangle inequality, $\omega(i) + d(i,j) + \omega(j) \leq \omega(i) + d(i,j) + d(j,r_i) + \omega(r_i) = \omega(i) + d(i,r_i) + \omega(r_i) \leq \lambda$. Therefore,

$$\omega(i) + d(i,j) + \omega(j) \leq \lambda \text{ iff } j \leq r_i. \quad (1)$$

Similarly, if $\ell_N \leq i$, then, using the node-triangle inequality, $\omega(i) + d(i,N) \leq \omega(\ell_N) + d(\ell_N,i) + d(i,N) = \omega(\ell_N) + d(\ell_N,N) \leq \lambda$. Therefore,

$$\omega(i) + d(i,N) \leq \lambda \text{ iff } \ell_N \leq i. \quad (2)$$
The algorithm computes all the indices \( r_i \), with \( 1 \leq i < N \), and the index \( \ell_N \). Since \( \omega(i) \geq \omega(i+1) - d(i, i+1) \), we have that \( r_i \leq r_{i+1} \). Therefore, all the \( r_i \)'s can be computed in \( O(N) \) time by scanning all the vertices of \( P \) in order from 1 to \( N \). Clearly, also \( \ell_N \) can be computed in \( O(N) \) time by scanning all the vertices of \( P \) in order from \( N \) downto 1. As \( d(1, N) > \lambda \), we have that \( r_1 < N \) and \( \ell_N > 1 \). We define the following two functions

\[
\bar{S}(i, j) := d(1, i) + c(i, j) + d(r_1 + 1, j) + \omega(r_1 + 1)
\]

and

\[
\bar{E}(i, j) := d(j, N) + c(i, j) + d(i, \ell_N - 1) + \omega(\ell_N - 1).
\]

Observe that both \( \bar{S}(i, j) \) and \( \bar{E}(i, j) \) can be computed in constant time. Moreover, using the graph-triangle inequality, we have that

- if \( r_1 < j \), then \( \bar{S}(i, j) \leq \bar{S}(i, j + 1) \);
- if \( i < \ell_N \), then \( \bar{E}(i, j) \leq \bar{E}(i + 1, j) \).

As the following lemma shows, the values \( \bar{S}(i, j) \) and \( \bar{E}(i, j) \) can be used to understand whether \( S(i, j) \leq \lambda \) and \( E(i, j) \leq \lambda \), respectively.

\begin{itemize}
  \item \textbf{Lemma 9.} If \( U(i, j) \leq \lambda \), then:
    \begin{itemize}
      \item \( S(i, j) \leq \lambda \) iff \( i \leq r_1 \) and \( \bar{S}(i, j) \leq \lambda \);
      \item \( E(i, j) \leq \lambda \) iff \( \ell_N \leq j \) and \( \bar{E}(i, j) \leq \lambda \).
    \end{itemize}
\end{itemize}

Let \( i < \sigma_i \leq N \) be the maximum index, or \( i \) if such an index does not exist, such that \( \bar{S}(i, \sigma_i) \leq \lambda \). Analogously, let \( 1 \leq \theta_j < j \) be the minimum index, or \( j \) if such an index does not exist, such that \( \bar{E}(\theta_j, j) \leq \lambda \). Our algorithm computes all the indices \( \sigma_i \), with \( 1 \leq i < N \), and the indices \( \theta_j \), with \( 1 < j \leq N \). By the graph-triangle inequality, \( \bar{S}(i, j) \leq \bar{S}(i + 1, j) \) as well as \( \bar{E}(i, j) \leq \bar{E}(i, j - 1) \). As a consequence, \( \sigma_{i+1} \leq \sigma_i \) and \( \theta_{j-1} \geq \theta_j \). Therefore, all the \( \sigma_i \)'s can be computed in \( O(N) \) time by scanning all the vertices of \( P \) in order from 1 to \( N \). Similarly, all the \( \theta_j \)'s can be computed in \( O(N) \) time by scanning all the vertices of \( P \) in order from \( N \) downto 1. The following lemma holds.

\begin{itemize}
  \item \textbf{Lemma 10.} If \( U(i, j) \leq \lambda \), then:
    \begin{itemize}
      \item \( S(i, j) \leq \lambda \) iff \( i \leq r_1 \) and \( j \leq \sigma_i \) (see also Figure 3);
      \item \( E(i, j) \leq \lambda \) iff \( \ell_N \leq j \) and \( \theta_j \leq i \) (see also Figure 3).
    \end{itemize}
\end{itemize}
Almost Optimal Algorithms for Diameter-Optimally Augmenting Trees

Let \( \rho_i \) be the minimum index, or \( \perp \) if such an index does not exist, such that \( \mu_i \leq \rho_i \leq \sigma_i \) and \( i \geq \theta_i \). The algorithm computes \( \rho_i \), for every \( i = 1, \ldots, N \). Since \( \mu_i \leq \mu_{i+1} \), \( \sigma_{i+1} \leq \sigma_i \), and \( \theta_{j-1} \geq \theta_j \), all the indices \( \rho_i \) can be computed in \( O(N) \) time. The following lemma holds.

\textbf{Lemma 11.} Let \( \langle P, \delta, \omega, c \rangle \) be an instance of WDOap, where \( c \) is a graph-metric and \( \omega \) satisfies the node-triangle inequality, and let \( \lambda > 0 \). There exists an index \( 1 \leq i < N \), such that \( \rho_i \) is defined and \( C(i, \rho_i) \leq \lambda \) iff the search version of WDOap on input instance \( \langle P, \delta, \omega, c, \lambda \rangle \) admits a feasible solution.

In the following we show how to check whether \( C(i, \rho_i) \leq \lambda \) in constant time after an \( O(N) \) time preprocessing. For every \( 1 \leq i < N \) such that \( r_i < N \), the algorithm computes

\[ \Delta(i) = \lambda - \omega(i) + d(i, r_i + 1) - \omega(r_i + 1). \]

Moreover, the algorithm computes \( \Delta_{\min} = \min_{1 \leq i < N} \Delta(i) \). Finally, for every \( i = 1, \ldots, N \) for which \( \rho_i \) is defined, our algorithm checks whether \( d(i, \rho_i) + c(i, \rho_i) \leq \Delta_{\min} \). If there exists \( i \) such that \( d(i, \rho_i) + c(i, \rho_i) \leq \Delta_{\min} \), then our algorithm returns \( (i, \rho_i) \). If this is not the case, then our algorithm returns \( \perp \). The following lemma proves the correctness of our algorithm.

\textbf{Lemma 12.} Let \( \langle P, \delta, \omega, c \rangle \) be an instance of WDOap, where \( c \) is a graph-metric and \( \omega \) satisfies the node-triangle inequality, and let \( \lambda > 0 \). The search version of WDOap on input instance \( \langle P, \delta, \omega, c, \lambda \rangle \) admits a feasible solution iff there exists an index \( 1 \leq i < N \), such that \( \rho_i \) is defined and \( d(i, \rho_i) + c(i, \rho_i) \leq \Delta_{\min} \).

We can conclude this section with the following theorem.

\textbf{Theorem 13.} Let \( \langle P, \delta, \omega, c \rangle \) be an instance of WDOap, where \( c \) is a graph-metric and \( \omega \) satisfies the node-triangle inequality, and let \( \lambda > 0 \). The search version of WDOap on input instance \( \langle P, \delta, \omega, c, \lambda \rangle \) can be solved in \( O(N) \) time and space.

\section{The algorithm for WDOap}

In this section we design an efficient \( O(N \log N) \) time and \( O(N) \) space algorithm that finds an optimal solution for instances \( \langle P, \delta, \omega, c \rangle \) of WDOap, where \( c \) is a graph-metric and \( \omega \) satisfies the node-triangle inequality. In the rest of the paper we denote by \( D^* \) the diameter of an optimal solution to the problem instance and, of course, we assume that \( D^* \) is not known by the algorithm. For the rest of this section, unless stated otherwise, \( i \) and \( j \) are two fixed indices such that \( 1 \leq i < j \leq N \). Similarly to the notation already used in the previous section, we define \( r_i \) as the maximum index such that \( i < r_i \leq N \) and \( \omega(i) + d(i, r_i) + \omega(r_i) \leq D^* \). If such an index does not exist, then \( r_i = i \). Analogously, we define \( \ell_N \) as the minimum index such that \( 1 \leq \ell_N < N \) and \( \omega(\ell_N) + d(\ell_N, N) \leq D^* \). If such an index does not exist, then \( \ell_N = N \). Our algorithm consists of the following three phases:

1. a precomputation phase in which the algorithm computes all the indices \( r_i \), with \( 1 \leq i < N \), and the index \( \ell_N \);  
2. a search-space reduction phase in which the algorithm reduces the size of the solution search space from \( \binom{N}{2} \) to \( N - 1 \) candidates;  
3. an optimal-solution selection phase in which the algorithm builds a data structure that is used to evaluate the leftover \( N - 1 \) solutions in \( O(\log N) \) time per solution.

Each of the three phases requires \( O(N \log N) \) time and \( O(N) \) space; furthermore, they all make use of the linear time algorithm for the search version of WDOap black-box. In the following we assume that \( d(1, N) > D^* \), as otherwise, any shortcut returned by our algorithm would be an optimal solution.
5.1 The precomputation phase

We perform a binary search over the indices from 1 to \( N \) and use the linear time algorithm for the search version of WDOAP to compute the maximum index \( \ell_N \) in \( O(N \log N) \) time and \( O(N) \) space. Indeed, when our binary search considers the index \( k \) as a possible choice of \( \ell_N \), it is enough to call the linear time algorithm for the search version of WDOAP with parameter \( \lambda = \omega(k + d(k, N)) \) and see whether the algorithm returns either a feasible solution or \( \bot \). Due to the node-triangle inequality, in the former case we know that \( \ell_N \leq k \), while in the latter case we know that \( \ell_N > k \).

Now we describe how to compute all the indices \( r_i \). Because of the node-triangle inequality \( r_i < N \) if \( i < \ell_N \). Therefore, we only have to describe how to compute \( r_i \) for every \( i < \ell_N \), since if \( i \geq \ell_N \), then \( r_i = N \). We use the linear time algorithm for the search version of WDOAP and perform a binary search over the set of sorted values \( \{ \omega(i) + d(i, i + 1) + \omega(i + 1) \mid 1 \leq i < \ell_N \} \) to compute the largest value of the set that is less than or equal to \( D^* \), if any. This allows us to compute, in \( O(N \log N) \) time and \( O(N) \) space, the set of all indices \( i < \ell_N \) for which \( r_i = i \). Now, for every index \( i < \ell_N \) for which \( r_i > i \), we set \( a_i = i + 1 \) and \( b_i = N \). Observe that \( a_i \leq r_i \leq b_i \). Next, using a two-round binary search, we restrict all the intervals \( [a_i, b_i] \)’s by updating both \( a_i \) and \( b_i \) while maintaining the invariant property \( a_i \leq r_i \leq b_i \) at the same time.

Let \( X \) be the set of indices \( i \), with \( 1 \leq i < \ell_N \), for which \( b_i \geq a_i + 2 \). The first round of the binary search ends exactly when \( X \) becomes empty. Each iteration of the first round works as follows. For every \( i \in X \), the algorithm computes the median index \( m_i = \lceil (a_i + b_i)/2 \rceil \). Now the algorithm computes the weighted median of the \( m_i \)’s, say \( m_\tau \), where the weight of \( m_i \) is equal to \( b_i - a_i \). Let

\[
X_\tau^+ = \{ i \in X \mid \omega(i) + d(i, m_i) + \omega(m_i) \geq \omega(\tau) + d(\tau, m_\tau) + \omega(m_\tau) \}
\]

and

\[
X_\tau^- = \{ i \in X \mid \omega(i) + d(i, m_i) + \omega(m_i) \leq \omega(\tau) + d(\tau, m_\tau) + \omega(m_\tau) \}.
\]

Observe that \( X = X_\tau^+ \cup X_\tau^- \) and \( \tau \in X^+, X^- \).

Now we call the linear time algorithm for the search version of WDOAP with parameter \( \lambda = \omega(\tau) + d(\tau, m_\tau) + \omega(m_\tau) \). If the algorithm finds two indices such that \( D(i, j) \leq \lambda \), then we know that \( D^* \leq \lambda \) and therefore, for every \( i \in X_\tau^+ \), we update \( b_i \) by setting it equal to \( m_i \). If the algorithm outputs \( \bot \), then we know that \( D^* > \lambda \) and therefore, for every \( i \in X_\tau^- \), we update \( a_i \) by setting it equal to \( m_i \). We observe that in either case, the invariant property \( a_i \leq r_i \leq b_i \) is kept because of (1). An iteration of the first round of the binary search ends right after the removal of all the indices \( i \) such that \( b_i = a_i + 1 \) from \( X \). Notice that, in the worst case, the overall sum of the intervals widths at the end of a single iteration is (almost) \( 3/4 \) times the same value computed at the beginning of the iteration. This implies that the first round of the binary search ends after \( O(\log N) \) iterations. Furthermore, the time complexity of each iteration is \( O(N) \). Therefore, the overall time needed for the first round of the binary search is \( O(N \log N) \).

The second round of the binary search works as follows. Because \( a_i \leq r_i \leq b_i \) and \( b_i \geq a_i + 1 \) for every \( i < \ell_N \) such that \( i < r_i \), we have that \( r_i \) is equal to either \( a_i \) or \( b_i \). To understand whether either \( r_i = a_i \) or \( r_i = b_i \), we sort the (at most) \( 2N \) values

\[
\Upsilon = \bigcup_{i < \ell_N, i < r_i} \{ \omega(i) + d(i, a_i) + \omega(a_i), \omega(i) + d(i, b_i) + \omega(b_i) \}
\]
and use binary search, together with the linear time algorithm for the search version of WDOAP, to compute the two consecutive distinct values $D^+, D^- \in \mathbb{T}$ such that $D^− < D^* \leq D^+$ (if $D^−$ does not exist, then we assume it to be equal to 0). Finally, we use the two values $D^+$ and $D^−$ to select either $a_i$ or $b_i$. More precisely, if $a_i = b_i$, then $r_i = a_i$. If $a_i \neq b_i$, then by the choice of $D^+$ and $D^−$, either $D^− < \omega(i) + d(i,a_i) + \omega(a_i) \leq D^+$ (i.e., $r_i = a_i$) or $D^− < \omega(i) + d(i,b_i) + \omega(b_i) \leq D^*$ (i.e., $r_i = b_i$). The time and space complexities of the second round of the binary search are $O(N \log N)$ and $O(N)$, respectively. We have proved the following lemma.

Lemma 14. The precomputation phase takes $O(N \log N)$ time and $O(N)$ space.

5.2 The search-space reduction phase

In the search-space reduction phase the algorithm computes a set of $N − 1$ candidates as optimal shortcut in $O(N \log N)$ time and $O(N)$ space. Let $f(i,j) := \max \{U(i,j), E(i,j)\}$. Since both $U(i,j)$ and $E(i,j)$ are monotonically non-increasing w.r.t. $j$, $f(i,j)$ is monotonically non-increasing w.r.t. $i$. For every $1 \leq i < N$, our algorithm computes the minimum index $1 < \psi_i \leq N$, if any, such that $f(i,\psi_i) \leq D^*$. As both $S(i,j)$ and $C(i,j)$ are monotonically non-decreasing w.r.t. $j$, it follows that the set $\{(i,\psi_i) \mid 1 \leq i < N\}$ contains an optimal solution to the problem instance.

We compute all the indices $\psi_i$’s using a two-round binary search technique similar to the one we already used in the precomputation phase. First, we set $a_i = i + 1$ and $b_i = N$, for every $1 \leq i < N$. Observe that $a_i \leq \psi_i \leq b_i$. In the two-round binary search, we restrict all the intervals $[a_i,b_i]$’s by updating both $a_i$ and $b_i$ while maintaining the invariant property $a_i \leq \psi_i \leq b_i$ at the same time.

Let $X$ be the set of indices $i$ for which $b_i \geq a_i + 2$. The first round of the binary search ends exactly when $X$ becomes empty. Each iteration of the first round works as follows. For every $i \in X$, the algorithm computes the median index $m_i = \lfloor (a_i + b_i)/2 \rfloor$. Next the algorithm computes the weighted median of the $m_i$’s, say $m_\tau$, where the weight of $m_i$ is equal to $b_i - a_i$. Let

$$X_+ = \{i \in X \mid f(i,m_i) \geq f(\tau,m_\tau)\} \quad \text{and} \quad X_- = \{i \in X \mid f(i,m_i) \leq f(\tau,m_\tau)\}.$$ 

Observe that $X = X_+ \cup X_-$; moreover, $\tau \in X_+, X_-$. Now we call the linear time algorithm for the search version of WDOAP with parameter $\lambda = f(\tau,m_\tau)$. If the algorithm finds two indices such that $D(i,j) \leq \lambda$, then we know that $D^* \leq f(\tau, m_\tau)$ and therefore, since $f(i,j) \leq f(i,j + 1)$, for every $i \in X_+$, we update $b_i$ by setting it equal to $m_i$. If the algorithm outputs $\bot$, then we know that $D^* > f(\tau, m_\tau)$ and therefore, by monotonicity of $f$, for every $i \in X_-$, we update $a_i$ by setting it equal to $m_i$. We observe that in either case, the invariant property $a_i \leq \psi_i \leq b_i$ is maintained. An iteration of the first round of the binary search ends right after the removal of all the indices $i$ such that $b_i = a_i + 1$ from $X$. Notice that, in the worst case, the overall sum of the intervals widths at the end of a single iteration is (almost) $3/4$ times the same value computed at the beginning of the iteration. This implies that the first round of the binary search ends after $O(\log N)$ iterations. Furthermore, both the time and space complexities of each iteration is $O(N)$. Therefore, the overall time needed for the first round of the binary search is $O(N \log N)$.

---

3 Observe that $E(i,j) = \max \{E(i,j), \omega(\xi_N) + d(\xi_N, N)\}$ because of the node-triangle inequality. However, since we know that $\omega(\xi_N) + d(\xi_N, N) \leq D^*$ by definition, we can check whether $E(i,j) \leq D^*$ by simply evaluating $E(i,j)$. 
The second round of the binary search works as follows. Because \( a_i \leq \psi_i \leq b_i \) and \( b_i \leq a_i + 1, \psi_i \) is either equal to \( a_i \) or to \( b_i \). To understand whether either \( \psi_i = a_i \) or \( \psi_i = b_i \), we sort the (at most) \( 2N \) values \( Y = \bigcup_{1 \leq i < N} \{ f(i, a_i), f(i, b_i) \} \) and use binary search, together with the linear time algorithm for the search version of WDOAP, to compute the two consecutive distinct values \( D^+, D^- \in Y \) such that \( D^- < D^* \leq D^+ \) (if \( D^- \) does not exist, then we assume it to be equal to 0). Finally, we use the two values \( D^+ \) and \( D^- \) to select either \( a_i \) or \( b_i \). More precisely, if \( a_i = b_i \), then \( \psi_i = a_i \). If \( a_i \neq b_i \), then by the choice of \( D^- \) and \( D^+ \), either \( D^- < f(i, a_i) \leq D^+ \) (i.e., \( \psi_i = a_i \)) or \( D^- < f(i, b_i) \leq D^+ \) (i.e., \( \psi_i = b_i \)).

The time and space complexities of the second round of the binary search are \( O(N \log N) \) and \( O(N) \), respectively. We have proved the following lemma.

\[ \textbf{Lemma 15.} \text{The search-space reduction phase takes } O(N \log N) \text{ time and } O(N) \text{ space. Furthermore, there exists a shortcut } (i^*, \psi_{i^*}) \text{ such that } D(i^*, \psi_{i^*}) = D^*. \]

### 5.3 The optimal-solution selection phase

In the optimal-solution selection phase, we build a data structure in \( O(N \log N) \) time and use it to evaluate the quality of the \( N - 1 \) candidates \( (1, \psi_1), \ldots, (N - 1, \psi_{N - 1}) \) in \( O(\log N) \) time per candidate. For every \( k = 1, \ldots, N \), we define

\[
\phi_k(x) := \omega(k) + \max \{ d(k, r_k) + \omega(r_k), x - d(k, r_k + 1) + \omega(r_k + 1) \}.
\]

Let \( \mathcal{U} := \max \{ \phi_k(x) \mid 1 \leq k < \ell_N \} \) be the upper envelope of all the functions \( \phi_k(x) \). Observe that each \( \phi_k(x) \) is itself the upper envelope of two linear functions. Therefore, \( \mathcal{U}(x) \) is the upper envelope of at most \( 2N - 2 \) linear functions. In [12] it is shown how to compute the upper envelope of a set of \( O(N) \) linear functions in \( O(N \log N) \) time and \( O(N) \) space. In the same paper it is also shown how the value \( \mathcal{U}(x) \) can be computed in \( O(\log N) \) time, for any \( x \in \mathbb{R} \).

We denote by \( x_i = d(i, \psi_i) + c(i, \psi_i) \) the overall weight of the edges of the unique cycle in \( P + (i, \psi_i) \). For every \( 1 \leq i < N \), we compute the value

\[
\eta_i = \max \{ \mathcal{U}(i, \psi_i), S(i, \psi_i), E(i, \psi_i), \mathcal{U}(x_i) \}. \tag{3}
\]

The algorithm computes the index \( \alpha \) that minimizes \( \eta_\alpha \) and returns the shortcut \( (\alpha, \psi_\alpha) \).

\[ \textbf{Lemma 16.} \text{For every } i, \text{ with } 1 \leq i < N \text{, } C(i, \psi_i) \leq \mathcal{U}(x_i). \]

Let \( i^* \) be the index such that \( D(i^*, \psi_{i^*}) = D^* \), whose existence is guaranteed by Lemma 15. The following lemma holds.

\[ \textbf{Lemma 17.} \mathcal{U}(x_{i^*}) \leq D^*. \]

We can finally conclude this section by stating the main results of this paper.

\[ \textbf{Theorem 18.} \text{WDOAP can be solved in } O(N \log N) \text{ time and } O(N) \text{ space.} \]

\[ \textbf{Theorem 19.} \text{DOAP on trees embedded in a (graph-)metric space can be solved in } O(n \log n) \text{ time and } O(n) \text{ space.} \]
References


Approximation Algorithms for Facial Cycles in Planar Embeddings

Giordano Da Lozzo
Computer Science Department, Roma Tre University, Italy
dalozzo@dia.uniroma3.it
https://orcid.org/0000-0003-2396-5174

Ignaz Rutter
Department of Computer Science and Mathematics, University of Passau, Germany
rutter@fim.uni-passau.de
https://orcid.org/0000-0002-3794-4406

Abstract

Consider the following combinatorial problem: Given a planar graph $G$ and a set of simple cycles $C$ in $G$, find a planar embedding $E$ of $G$ such that the number of cycles in $C$ that bound a face in $E$ is maximized. This problem, called MAX FACIAL C-CYCLES, was first studied by Mutzel and Weiskircher [IPCO ‘99] and then proved NP-hard by Woeginger [Oper. Res. Lett., 2002].

We establish a tight border of tractability for MAX FACIAL C-CYCLES in biconnected planar graphs by giving conditions under which the problem is NP-hard and showing that strengthening any of these conditions makes the problem polynomial-time solvable. Our main results are approximation algorithms for MAX FACIAL C-CYCLES. Namely, we give a 2-approximation for series-parallel graphs and a $(4 + \varepsilon)$-approximation for biconnected planar graphs. Remarkably, this provides one of the first approximation algorithms for constrained embedding problems.

2012 ACM Subject Classification Mathematics of computing → Graph theory

Keywords and phrases Planar Embeddings, Facial Cycles, Complexity, Approximation Algorithms

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.41

Related Version A full version of the paper containing omitted or sketched proofs is available as [7], https://arxiv.org/abs/1607.02347.

Acknowledgements This research was partially supported by MIUR Project “MODE”, by H2020-MSCA-RISE project “CONNECT”, and by MIUR-DAAD JMP N° 34120.

1 Introduction

A planar graph is a graph that can be embedded into the plane, i.e., it can be drawn into the plane without crossings. Such an embedding partitions the plane into topologically connected regions, called faces. There is exactly one unbounded face, which is called outer face. While there exist infinitely many such embeddings, the embeddings for connected graphs can be grouped into finitely many equivalence classes of combinatorial embeddings, where two embeddings are equivalent if the clockwise cyclic order of the edges around each vertex is the same and their outer face is bounded by the same walk. Since a graph may admit exponentially many different such embeddings, several drawing algorithms for planar graphs simply assume that one embedding has been fixed beforehand and draw the graph with this fixed embedding. Often, however, the quality of the resulting drawing depends...
Approximation Algorithms for Facial Cycles in Planar Embeddings

strongly on this embedding; examples are the number of bends in orthogonal drawings [17] or the area requirement of planar straight-line drawings [8].

Consequently, there is a long line of research that seeks to optimize quality measures over all combinatorial embeddings. Not surprisingly, except for a few notable cases such as minimizing the radius of the dual graph [2, 15], many of these problems have turned out to be NP-complete. For example it is NP-complete to decide whether there exists a planar embedding that allows for a planar orthogonal drawing without bends or for an upward planar drawing [12]. While there has been quite a bit of work on solving these problems for special cases, e.g., for the orthogonal bend minimization problem [4, 5], to the best of our knowledge, approximation algorithms have rarely been considered.

Another way of describing a combinatorial embedding of a connected graph $G$ is by describing its facial walks, i.e., by listing the walks of $G$ that bound a face. In the case of biconnected planar graphs, the facial walks are simple, and we refer to them as facial cycles. In this paper, we consider the problem of optimizing the set of facial cycles. Given a list $C$ of cycles in a biconnected graph $G$, the problem $\text{Max Facial } C$-Cycles asks for an embedding $E$ of $G$ such that the number of cycles in $C$ that are facial cycles of $E$ is maximized. The practical motivation for this problem comes from the need to visualize graphs in such a way that particular substructures, in this case cycles, are clearly recognizable. These structures may be either provided manually by the user or be the result of an automated analysis. We note that, given a biconnected planar graph $G$ and a set $C$ of cycles of $G$, it can be efficiently decided whether there exists a planar embedding of $G$ in which all cycles of $C$ are facial cycles. For each cycle $C \in C$, we subdivide each edge of $C$ once and connect the subdivision vertex to a new vertex $v_C$. If the resulting graph is planar, then the desired embedding of $G$ can be obtained by removing all vertices $v_C$ and their incident edges. However, from a practical point of view, this approach is insufficient, as it does not produce a solution if it is not possible to simultaneously have all the cycles in $C$ as facial cycles. Instead, we would like to compute an embedding that maximizes the number of cycles in $C$ that are facial cycles.

The research on this problem was initiated by Mutzel and Weiskircher [16], who gave an integer linear program (ILP) for a weighted version. Woeginger [19] showed that the problem is NP-complete by showing that it is NP-complete to maximize the number of facial cycles that have size at most 4. Da Lozzo et al. [6] consider the problem of deciding whether there exists an embedding such that the maximum face size is $k$. They give polynomial-time algorithms for $k \leq 4$, show NP-hardness for $k \geq 5$, and give a factor-6 approximation for minimizing the size of the largest face. Dornheim [10] studies a decision problem subject to so-called topological constraints, which specify for certain cycles of a planar graph two subsets of edges of the graph that have to be embedded inside and outside the respective cycle; note that a cycle is a facial cycle if its interior is empty. He proved NP-completeness and reduced the connected case to the biconnected case. Another related problem, known as Partially Embedded Planarity (for short PEP), has been studied by Angelini et al. [1]. Given a planar graph $G$ and an embedding $\mathcal{E}_H$ of a subgraph $H$ of $G$, the PEP problem asks for the existence of an embedding $\mathcal{E}_G$ of $G$ that extends $\mathcal{E}_H$, i.e., the restriction of $\mathcal{E}_G$ to $H$ coincides with $\mathcal{E}_H$. As observed before, if $H$ is biconnected, then its combinatorial embedding is fully specified by the set of cycles of $H$ (of $G$) that are faces in $\mathcal{E}_H$. Thus, the $\text{Max Facial } C$-Cycles problem can be interpreted as an optimization counterpart of PEP in which one tries to minimize the set of faces of $H$ that are not faces in the final embedding of $G$.

Contribution and outline. We thoroughly study the complexity of $\text{Max Facial } C$-Cycles for biconnected planar graphs. We start with preliminaries concerning connectivity and the
SPQR-tree data structure in Section 2. In Section 3, we show that MAX FACIAL C-CYCLES is NP-complete even if each cycle in $C$ intersects any other cycle in $C$ in at most two vertices and intersects at most three other cycles of $C$. In Section 4, we complement these results with efficient algorithms for series-parallel and general planar graphs when the cycles intersect only few other cycles in more than one vertex. We note that, though these instances are fairly restricted, this establishes a tight border of tractability for the problem in the sense that dropping or strengthening any of the conditions for our algorithms yields an NP-hard problem. Moreover, the techniques for obtaining these results are the basis for our main result in Section 5, where we develop efficient approximation algorithms for the problem. More specifically, we give a 2-approximation for series-parallel graphs and a $(4 + \varepsilon)$-approximation for biconnected planar graphs, where $\varepsilon > 0$ is a constant. We remark that, to the best of our knowledge, this work and our contribution in [6] provide one of the very few known approximability results concerning constrained combinatorial embeddings.

A full version of the paper containing omitted or sketched proofs is available as [7].

2 Connectivity and SPQR-trees

A graph $G$ is connected if there is a path between any two vertices. A cutvertex (separating pair) is a vertex (a pair of vertices) whose removal disconnects the graph. A connected (biconnected) graph is biconnected (triconnected) if it does not have a cutvertex (a separating pair).

We consider $uv$-graphs with two special pole vertices $u$ and $v$, which can be recursively defined as follows. An edge $uv$ is an $uv$-graph with poles $u$ and $v$. Now let $G_i$ be an $uv$-graph with poles $u_i$ and $v_i$, for $i = 1, \ldots, k$, and let $H$ be a planar graph with two designated vertices $u$ and $v$ and $k + 1$ edges $uv, e_1, \ldots, e_k$. We call $H$ the skeleton of the composition and its edges are called virtual edges; the edge $uv$ is the parent edge and $u$ and $v$ are the poles of the skeleton $H$. To compose the $G_i$ into an $uv$-graph with poles $u$ and $v$, we remove the edge $uv$ and replace each $e_i$ by $G_i$, for $i = 1, \ldots, k$, by removing $e_i$ and identifying the poles of $G_i$ with the endpoints of $e_i$. In fact, we only allow three types of compositions: in a series composition the skeleton $H$ is a cycle of length $k + 1$, in a parallel composition $H$ consists of two vertices connected by $k + 1$ parallel edge, and in a rigid composition $H$ is triconnected.

It is known that for every biconnected graph $G$ with an edge $uv$ the graph $G - uv$ is an $uv$-graph with poles $u$ and $v$. The $uv$-graph $G - uv$ gives rise to a (de-)composition tree $T$ describing how it can be obtained from single edges. Refer to Fig. 1. The nodes of $T$ corresponding to edges, series, parallel, and rigid compositions of the graph are $Q$-, $S$-, $P$-, and $R$-nodes, respectively. To obtain a composition tree for $G$, we add an additional root Q-node representing the edge $uv$. To fully describe the composition, we associate with each node $\mu$ its skeleton denoted by skel($\mu$). For a node $\mu$ of $T$, the pertinent graph pert($\mu$) is the subgraph represented by the subtree with root $\mu$. Similarly, for a virtual edge $\varepsilon$ of a skeleton skel($\mu$), the expansion graph of $\varepsilon$, denoted by exp($\varepsilon$), is the pertinent graph pert($\mu'$) of the neighbour $\mu'$ of $\mu$ corresponding to $\varepsilon$ when considering $T$ rooted at $\mu$.

The SPQR-tree of $G$ with respect to the edge $uv$, originally introduced by Di Battista and Tamassia [9], is the (unique) smallest decomposition tree $T$ for $G$. Using a different edge $u'v'$ of $G$ and a composition of $G - u'v'$ corresponds to rerooting $T$ at the node representing $u'v'$. It thus makes sense to say that $T$ is the SPQR-tree of $G$. The SPQR-tree of $G$ has size linear in $G$ and can be computed in linear time [13]. Planar embeddings of $G$ correspond bijectively to planar embeddings of all skeletons of $T$; the choices are the orderings of the parallel edges in $P$-nodes and the embeddings of the $R$-node skeletons, which are unique up
Approximation Algorithms for Facial Cycles in Planar Embeddings

Figure 1: (left) A biconnected planar graph $G$ and (right) the SPQR-tree $T$ of $G$ rooted at edge $e = uv$. The skeletons of all non-leaf nodes of $T$ are depicted; virtual edges corresponding to edges of $G$ are thin, whereas virtual edges corresponding to S-, P-, and R-nodes are thick. Dashed arrowed curves connect the (dotted) parent edge in the skeleton of a child node with the virtual edge representing the child node in the skeleton of its parent.

to a flip. When considering rooted SPQR-trees, we assume that the embedding of $G$ is such that the root edge is incident to the outer face, which is equivalent to the parent edge being incident to the outer face in each skeleton. We remark that in a planar embedding of $G$, the poles of any node $\mu$ of $T$ are incident to the outer face of $\text{pert}(\mu)$. Hence, in the following we only consider such embeddings.

Let $\mu$ be a node of $T$ with poles $u$ and $v$. We assume that edge $uv$ is part of $\text{skel}(\mu)$ and $\text{pert}(\mu)$. Note that, due to this addition, $\text{pert}(\mu)$ may not be a subgraph of $G$ anymore. The outer face of a embedding of $\text{pert}(\mu)$ is the one obtained from such an embedding after removing the edge $(u, v)$ connecting its poles.

3 Complexity

In this section we study the computational complexity of the underlying decision problem FACIAL $C$-CYCLES of MAX FACIAL $C$-CYCLES, which given a biconnected planar graph $G$, a set $C$ of simple cycles of $G$, and a positive integer $k \leq |C|$, asks whether there exists a planar embedding $E$ of $G$ such that at least $k$ cycles in $C$ are facial cycles of $E$. FACIAL $C$-CYCLES is in NP, as we can guess a set $C' \subseteq C$ of $k$ cycles and then check in polynomial time (in $|G| + |C|$) whether an embedding of $G$ exists in which all cycles in $C'$ are facial. We show NP-hardness for general graphs and for series-parallel graphs.

\textbf{Theorem 1.} FACIAL $C$-CYCLES is NP-complete, even if each cycle $C \in C$

- intersects any other cycle in $C$ in at most two vertices, and
- intersects at most three other cycles of $C$ in more than one vertex.

\textbf{Proof sketch.} We give a reduction from MAXIMUM INDEPENDENT SET in triconnected cubic planar graphs, which is NP-complete [14]. Let $H$ be a triconnected cubic planar graph. Observe that $H$ has a unique combinatorial embedding up to a flip [18]. We construct an instance $(G, C, k)$ of FACIAL $C$-CYCLES as follows; see Fig. 2. Take the planar dual $H^*$ of $H$, which is a triangulation, and take $C$ as the set of facial cycles of $H^*$. The graph $G$ is obtained from $H^*$ by adding for each edge $e = uv \in E(H^*)$ an edge vertex $v_e$ with neighbors $u$ and $v$. It is not hard to see that $H$ admits an independent set of size $k$ if and only if $G$ admits a combinatorial embedding where $k$ cycles in $C$ are facial (see [7]).
In this section we discuss special cases of Max Facial C-Cycles that admit a polynomial-time solution. In particular, we show that strengthening any of the conditions in Theorem 1...
Approximation Algorithms for Facial Cycles in Planar Embeddings

Figure 4 (b) Skeleton of the R-node $\mu$ of the SPQR-tree $T$ rooted at edge $e = uv$ of the graph depicted in (a); see Fig. 1 for an illustration of tree $T$. The green and red cycles in (a) are relevant for $\mu$ as they project to the green and red cycle in (b), respectively. The red cycle is an interface cycle. Dotted edges and dotted virtual edges are associated with the parent of $\mu$. (c) Pertinent graph of the P-node depicted in (d) with three virtual edges corresponding to children from left to right realizing none, the green and the red, and only the red cycle, respectively. The red cycle bounds face $f$ in (c) since, in addition, the second and third child are adjacent in the embedding of the skeleton.

or Theorem 2 makes the problem tractable.

4.1 General Planar Graphs

We study Max Facial $\mathcal{C}$-Cycles when each cycle in $\mathcal{C}$ intersects at most two other cycles in $\mathcal{C}$ in more than one vertex. In this setting, we give in Theorem 9 a quadratic-time algorithm for biconnected planar graphs. For series-parallel graphs we present in Theorem 10 an FPT algorithm with respect to the maximum number of cycles in $\mathcal{C}$ sharing two or more vertices with any cycle in $\mathcal{C}$. We remark that our algorithms imply that strengthening any of the two conditions of Theorem 1 results in a polynomial-time solvable problem. In particular, Max Facial $\mathcal{C}$-Cycles is polynomial-time solvable if any two cycles in $\mathcal{C}$ share at most one vertex.

We compute the optimal solution in these cases by a dynamic program that works bottom-up in the SPQR-tree $T$ of $G$. Let $\mu$ be a node of $T$. We call a cycle $C \in \mathcal{C}$ relevant for $\mu$ (or for $\text{skel}(\mu)$) if it projects to a cycle in $\text{skel}(\mu)$, that is, the vertices of $C$ in $\text{skel}(\mu)$ and the edges of $\text{skel}(\mu)$ that contain vertices or edges in $C$ form a cycle $C'$ in $\text{skel}(\mu)$ with at least two edges. The cycle $C'$ is the projection of the cycle $C$ in $\text{skel}(\mu)$. Similarly, we also define the projection of a cycle $C \in \mathcal{C}$ to $\text{pert}(\mu)$. The cycle $C$ is an interface cycle of $\mu$ if its projection $C'$ contains the parent edge of $\text{skel}(\mu)$. Refer to Figs. 4a and 4b. We denote the set of relevant cycles and of interface cycles of a node $\mu$ by $R(\mu)$ and by $I(\mu)$, respectively. Clearly, $I(\mu) \subseteq R(\mu)$. We denote $I(\mu) = \{X \subseteq I(\mu) \mid |X| \leq 2\}$ as the set of possible interfaces. Let $\mu$ be a node of $T$. We have the following two important observations.

\begin{itemize}
  \item \textbf{Observation 3.} If each cycle in $\mathcal{C}$ intersects at most two other cycles in $\mathcal{C}$ in more than one vertex, then $|I(\mu)| \leq 3$.
  \item \textbf{Observation 4.} In any combinatorial embedding $\mathcal{E}$ of $G$ at most two interface cycles of $\mu$ can simultaneously bound a face in $\mathcal{E}$.
\end{itemize}

Observation 3 holds since all interface cycles of a node $\mu$ share at least the poles of $\mu$. Observation 4 holds since each interface cycle can only bound one of the two faces incident to the virtual edge representing the parent of $\mu$ in $\text{skel}(\mu)$.

Thus, to the rest of $G$, the only relevant information about a combinatorial embedding $\mathcal{E}_\mu$ of $\text{pert}(\mu)$ is

(a) the number of facial cycles in $\mathcal{C}$ that bound a face of $\mathcal{E}_\mu$ and
(b) the set of cycles in \( C \) that project to the facial cycles incident to the parent edge of \( \pert(\mu) \).

The reason for (a) is that cycles of \( C \) that are facial cycles of \( E_\mu \) not incident to the parent edge will be facial cycles of any embedding of \( G \) where the embedding of \( \pert(\mu) \) is \( E_\mu \). So it suffices to track their number rather than which cycles are facial. For (b) observe that only those cycles that project to a face incident to \( E_\mu \) can potentially be realized by any embedding of \( G \) where the embedding of \( \pert(\mu) \) is \( E_\mu \). We thus have to keep track of them. However, by Observation 4, at most two of them can eventually become facial cycles, and hence it suffices to consider any combination of at most two cycles for this interface.

If \( E \) is a combinatorial embedding of \( \pert(\mu) \) and the elements of \( I \in I(\mu) \) project to distinct faces incident to the parent edge in \( \pert(\mu) \), we say that \( E \) realizes \( I \); see Figs. 4c and 4d.

For any node \( \mu \) and any set \( I \in I(\mu) \), we denote by \( T[\mu, I] \) the maximum number \( k \) such that there exists a combinatorial embedding \( E \) of \( \pert(\mu) \) that realizes \( I \) and such that \( k \) cycles in \( C \) bound a face of \( E \) that is not incident to the parent edge of \( \pert(\mu) \). If no such embedding exists, we set \( T[\mu, I] = -\infty \). Due to Observation 4, for convenience we extend the definition of \( T \) to the case in which the size of \( I \) is larger than 2; in this case, we define \( T[\mu, I] = -\infty \).

We show how to compute the entries of \( T \) in a bottom-up fashion in the SPQR-tree \( T \) of \( G \). It is not hard to modify the dynamic program to additionally output a corresponding combinatorial embedding of \( G \). We root \( T \) at an arbitrary Q-node \( \rho \). Let \( \phi \) be the unique child of \( \rho \). Note that the maximum number of facial cycles in \( C \) for any combinatorial embedding of \( G \) is \( \max_{I \in I(\emptyset)} |I| + T[\phi, I] \) for any leaf Q-node \( \mu \), we have that \( T[\mu, I] = 0 \) for each \( I \in I(\mu) \). The following lemmata deal with the different types of inner nodes in an SPQR-tree.

**Lemma 5.** Let \( \mu \) be an S-node with children \( \mu_i, i = 1, \ldots, k \). Then, \( T[\mu, I] = \sum_{i=1}^{k} T[\mu_i, I] \), for \( I \in I(\mu) \). Also, each entry \( T[\mu, I] \) can be computed in \( O(k) \) time.

**Proof.** The lemma follows easily from the observation that a combinatorial embedding of \( \pert(\mu) \) realizes \( I \) if and only if each of its children realizes \( I \).

**Lemma 6.** Let \( \mu \) be a P-node with children \( \mu_1, \ldots, \mu_k \). Then

\[
T[\mu, I] = \max_{I \subseteq C \subseteq R(\mu)} \left( \sum_{i=1}^{k} T[\mu_i, C_{\mu_i}] + f(C) \right),
\]

where (i) \( C_{\mu_i} = C \cap I(\mu_i) \) and (ii) \( f(C) = |C \setminus I| \) if \( \skel(\mu) \) admits a planar embedding \( E \) such that (a) each two virtual edges \( e_i \) and \( e_j \) corresponding to children \( \mu_i \) and \( \mu_j \) of \( \mu \), respectively, such that \( |C_{\mu_i} \cap C_{\mu_j}| = 1 \) are adjacent in \( E \), and where (b) the virtual edges \( e' \) and \( e'' \) corresponding to the children \( \mu' \) and \( \mu'' \) of \( \mu \) such that \( C_{\mu'} \cap I \neq \emptyset \) and \( C_{\mu''} \cap I \neq \emptyset \), respectively, are incident to the outer face of \( E \), and \( f(C) = -\infty \) otherwise.

**Proof.** Consider an embedding of \( \pert(\mu) \) that embeds \( T[\mu, I] \) cycles of \( C \) as facial cycles and the corresponding embedding \( E \) of \( \skel(\mu) \). Let \( C \subseteq R(\mu) \) denote the set of cycles in \( C \) that are facial cycles in \( E \) or that are in \( I \). Obviously, to make a cycle \( c \in C \setminus I \) a facial cycle, each of the two children of \( \mu \) that contain \( c \) in their interface (i) must be adjacent in \( E \) and (ii) must both realize cycle \( c \). Also, in order for the cycles in \( I \) to bound the outer face of the embedding of \( \pert(\mu) \), the two children of \( \mu \) containing such interface cycles (i) must be incident to the outer face of \( E \) and (ii) must each realize one of these cycles in their interface. Hence \( T[\mu, C] \) is a lower bound on the number of facial cycles in \( C \) in the embedding of \( \pert(\mu) \). On the other hand, it is not hard to see that by picking the maximum over all subsets \( C \subseteq R(\mu) \) this bound is attained.
We note that the existence of a corresponding embedding for a P-node $\mu$ with $k$ children can be tested in $O(k)$ time for any set $C \subseteq R(\mu)$, thus allowing us to evaluate $f(C)$ efficiently as follows. Consider the auxiliary multigraph $O$ that contains a vertex for each virtual edge of $\text{skel}(\mu)$, except for the edge representing the parent of $\mu$, and two such edges are adjacent if and only if there is a cycle in $C \setminus I$ that contains edges from both expansion graphs. Also, if there exist two virtual edges in $\text{skel}(\mu)$ containing edges from cycles in $I$, multigraph $O$ contains an edge between them. A corresponding embedding exists if and only if $O$ is either a simple cycle or it is a collection of paths. In latter case, $O$ can be augmented to a simple cycle and the order of the virtual edges along this cycle defines a suitable embedding of $\text{skel}(\mu)$.

Generally, the size of $R(\mu)$ can be large. However, if every cycle $C \in C$ shares two or more vertices with at most $r$ other cycles in $C$, the running time can be bounded as follows.

**Lemma 7.** Let $\mu$ be a P-node with children $\mu_1, \ldots, \mu_k$ such that any cycle of $R(\mu)$ shares two or more vertices with at most $r$ other cycles in $R(\mu)$. For each set $I \in I(\mu)$, table $T[\mu, I]$ can be computed in $O(r^2 \cdot n \cdot 2^r \cdot k)$ time from $T[\mu_i, I]$ with $i = 1, \ldots, k$.

**Proof.** We employ Lemma 6. It is $|R(\mu)| \leq r + 1$, and $|I(\mu)| = O(r^2)$. For each $I \in I(\mu)$ we need to consider all the sets $C \subseteq R(\mu)$ such that $I \subseteq C$. There are $O(2^r)$ such sets $C$ and for each of them we evaluate $f(C)$ in $O(k)$ time. ▶

We now deal with R-nodes. Let $\mu$ be an R-node. Note that the instance in the hardness of Theorem 1 is an R-node whose children are a parallel of an edge and a path of length 2. If, however, any cycle in $C$ shares two or more vertices with at most two other cycles from $C$, then the subgraph of the dual of $\text{skel}(\mu)$ induced by the faces that are projections of cycles in $R(\mu)$ consists of paths and cycles. We exploit the fact that these graphs have maximum degree 2 to give an efficient algorithm via dynamic programming.

**Lemma 8.** Let $\mu$ be an R-node with children $\mu_1, \ldots, \mu_k$. There is an $O(k^2)$-time algorithm for computing $T[\mu, I]$ from $T[\mu_i, I]$ for $i = 1, \ldots, k$, provided that cycles in $C$ share two or more vertices with at most two other cycles from $C$.

Altogether, Lemmas 5, 7, and 8 imply the following theorem.

**Theorem 9.** Max Facial $C$-Cycles can be solved in $O(n^2)$ time if every cycle in $C$ intersects at most two other cycles in more than one vertex.

### 4.2 Series-Parallel Graphs

In this section we consider Max Facial $C$-Cycles on series-parallel graphs. Combining the results from Lemma 5 and Lemma 7 yields the following.

**Theorem 10.** Max Facial $C$-Cycles is solvable in $O(r^2 \cdot n \cdot 2^r)$ time for series-parallel graphs if any cycle in $C$ intersects at most $r$ other cycles in two or more vertices.

**Corollary 11.** Max Facial $C$-Cycles is solvable in $O(n)$ time for series-parallel graphs if any cycle in $C$ intersects at most two other cycles.

In the following we show that Max Facial $C$-Cycles can be solved in polynomial time for series-parallel graphs if any two cycles in $C$ share at most two vertices. The next lemma shows the special structure of relevant cycles in P-nodes of the SPQR-tree in this case.

**Lemma 12.** Let $G$ be a series-parallel graph and $C$ be a set of cycles in $G$ such that any two cycles share at most two vertices. For each P-node $\mu$ any two cycles in $C$ that are relevant for $\mu$ are either edge-disjoint in $\text{skel}(\mu)$ or they share the unique virtual edge of $\text{skel}(\mu)$ that corresponds to a Q-node child of $\mu$, if any.
Then, \( \tilde{T} \) of two cycles in \( \mu \) within a factor of 2, for R-nodes, we achieve an approximation ratio of 5. We first show that Lemma 14. This time, however, instead of computing parallel graphs and in biconnected planar graphs. Again, we use dynamic programming on the SPQR-tree. This time, however, instead of computing parallel graphs and in biconnected planar graphs. Again, we use dynamic programming on the SPQR-tree of a series-parallel graph to obtain a \( \tilde{T} \) of it. A table is contained in both \( C \) and \( C' \), which is contained in both \( C \) and \( C' \), a contradiction. Further observe that a P-node may have at most one child that is a Q-node. This concludes the proof. ▶

We again use a bottom-up traversal of the SPQR-tree of a series-parallel graph to obtain the following theorem. The S-nodes are handled using Lemma 5 and the structural properties guaranteed by Lemma 12 allow for a simple handling of the P-nodes.

▶ Theorem 13. Max Facial C-Cycles is solvable in \( O(n) \) for series-parallel graphs if any two cycles in \( C \) share at most two vertices.

5 Approximation Algorithms

In this section we derive constant-factor approximations for Max Facial C-Cycles in series-parallel graphs and in biconnected planar graphs. Again, we use dynamic programming on the SPQR-tree. This time, however, instead of computing \( \tilde{T} \) of it, we compute an approximate version \( \tilde{T}(\mu, I) \) of it. A table \( \tilde{T}(\mu, I) \) is a \( c \)-approximation of \( T(\mu, I) \) if \( 1/c \cdot T(\mu, I) \leq \tilde{T}(\mu, I) \leq T(\mu, I) \), for all \( I \in I(\mu) \). For P-nodes, we give an algorithm that approximates each entry within a factor of 2, for R-nodes, we achieve an approximation ratio of \( (4 + \varepsilon) \) for any \( \varepsilon > 0 \). In the following lemmas we deal separately with S-, P-, and R-nodes.

▶ Lemma 14. Let \( \mu \) be an S-node with children \( \mu_1, \ldots, \mu_k \) and let \( \tilde{T}(\mu_i, I) \) be a \( c \)-approximation of \( T(\mu_i, I) \) for \( i = 1, \ldots, k \). Then, \( \tilde{T}(\mu, I) = \sum_{i=1}^{k} \tilde{T}(\mu_i, I) \) is a \( c \)-approximation of \( T(\mu, I) \).

Proof. To see this, observe that by Lemma 5, it is \( 1/c \cdot T(\mu, I) = 1/c \cdot \sum_{i=1}^{k} T(\mu_i, I) \leq \sum_{i=1}^{k} \tilde{T}(\mu_i, I) \) and \( \sum_{i=1}^{k} \tilde{T}(\mu_i, I) \leq \sum_{i=1}^{k} T(\mu_i, I) = T(\mu, I) \). ▶

Next we deal with a P-node \( \mu \) with children \( \mu_1, \ldots, \mu_k \). The algorithm works as follows. Fix a set \( I \in I(\mu) \). We construct an auxiliary graph \( H \) as follows. The vertices of \( H \) are the children \( \mu_1, \ldots, \mu_k \) of \( \mu \). Two vertices \( \mu_i \) and \( \mu_j \) are adjacent in \( H \) if and only if there exists a cycle \( C \in \mathcal{C} \) that intersects \( \mu_i \) and \( \mu_j \) such that \( \tilde{T}(\mu_x, (I \cup \{C\}) \cap I(\mu_x)) = \tilde{T}(\mu_x, I \cap I(\mu_x)) \) for \( x \in \{i, j\} \), i.e., according to the approximate table \( \tilde{T} \) additionally realizing \( C \) in the interface of the children \( \mu_i \) and \( \mu_j \) does not decrease the number of facial cycles of \( \text{pert}(\mu) \) in \( C \). If \( |I| = 2 \), assume that \( \mu_1 \) and \( \mu_2 \) are the two children intersected by the cycles in \( I \). Unless \( \mu_1 \) and \( \mu_2 \) are the only children of \( \mu \), we remove the edge \( \mu_1\mu_2 \) from \( H \) if it is there. This reflects the fact that, due to the restrictions imposed by \( I \), it is not possible to realize a corresponding cycle. Now compute a maximum matching \( M \) in \( H \). The matching \( M \) corresponds to a set \( \mathcal{C}_M \subseteq \mathcal{R}(\mu) \) of relevant cycles of \( \mu \) that are pairwise edge-disjoint. We set \( \tilde{T}(\mu, I) = \sum_{C \in \mathcal{C}_M} \tilde{T}(\mu, I \cap \mu_x) + |M| = \sum_{C \in \mathcal{C}_M} \tilde{T}(\mu, I \cap \mu_x) + |M| \). We claim that this gives a max\{2, c\}-approximation of \( T(\mu, I) \) if the \( \tilde{T}(\mu_i, I) \) are \( c \)-approximations of \( T(\mu_i, I) \).

▶ Lemma 15. Let \( \mu \) be a P-node an let \( \tilde{T}(\mu, I) \) be the table computed in the above fashion. Then, \( \tilde{T}(\mu, I) \) is a max\{2, c\}-approximation of \( T(\mu, I) \) if \( \tilde{T}(\mu_i, I) \) is a \( c \)-approximation of \( T(\mu_i, I) \).

Proof. We first show that \( \tilde{T}(\mu, I) \leq T(\mu, I) \). To this end, it suffices to show that, for any \( I \in I(\mu) \), there exists an embedding of \( \text{pert}(\mu) \) that realizes \( I \) and has \( \tilde{T}(\mu, I) \) facial.
cycles in $C$. Consider the multigraph with vertex set $\{\mu_1, \ldots, \mu_k\}$ and edge set $C_M \cup I$. This graph has maximum degree 2 and, due to the special treatment of the edge $\mu_1\mu_2$, unless $k = 2$, none of its connected components is a cycle. We can thus always complete this graph into a cycle containing all $\mu_i$, which defines a circular order of $\mu_1, \ldots, \mu_k$, and hence an embedding of $skel(\mu)$. In this embedding, all the cycles in $C_M \cup I$ project to facial cycles. Realizing all these cycles yields $\tilde{T}[\mu, I] = \sum_{i=1}^{k} T[\mu_i, (I \cup C_M) \cap I(\mu_i)] + |M| \leq \sum_{i=1}^{k} T[\mu_i, (I \cup C_M) \cap I(\mu_i)] + |M|$ realized cycles. By the definition of the $T[\mu_i, \cdot]$ we get embeddings for the pert($\mu_i$) with a corresponding number of cycles in $C$. Combining them according to the embedding of $skel(\mu)$ from above, yields an embedding of pert($\mu$) that realizes $I$ and has at least $\tilde{T}[\mu, I]$ facial cycles in $C$. Hence $\tilde{T}[\mu, I] \leq T[\mu, I]$.

Conversely, consider $T[\mu, I]$ and a corresponding embedding of $skel(\mu)$. Denote by $C_{opt}$ the set of facial cycles in $C$ in an optimal solution that project to facial cycles of $skel(\mu)$. We consider two cycles in $C_{opt}$ as adjacent if they intersect the same child of $\mu$. Clearly, each child $\mu_i$ is intersected by at most two cycles in $C_{opt}$ and, moreover, the two faces of $skel(\mu)$ incident to the parent edge are not realized. Hence the corresponding graph is a collection of paths, and it can be edge-colored with two colors. Let $C'_{opt}$ be the cycles in the larger color class. We have $|C'_{opt}| \geq |C_{opt}|/2$ and no two distinct cycles in $C'_{opt}$ intersect the same child $\mu_i$ of $\mu$, i.e., interpreting the cycles in $C'_{opt}$ as edges on the vertex set $\{\mu_1, \ldots, \mu_k\}$ yields a matching $M'$. We would like to argue that our matching $M$ in the auxiliary graph $H$ is larger than $M'$, and hence we realize at least half of the cycles of the optimum. However, this argument is not valid, since $M'$ may contain edges that are not present in $H$ due to approximation errors in the $T[\mu_i, \cdot]$. We will show that the contribution of these edges is irrelevant and hence the intuition about comparing the matching sizes indeed applies.

Let $M'_1 = M' \setminus E(H)$ and $M'_2 = M' \cap E(H)$. Let $J = \{1, \ldots, k\}$ and let $J_1 = \{i \in J \mid \exists C \in M'_1$ that intersects $\mu_i\}$ be the indices of children that are intersected by a cycle in $M'_1$. The set $J_2 = J \setminus J_1$ contains the remaining indices.

Clearly, we have $T[\mu, I] = \sum_{i=1}^{k} T[\mu_i, (I \cup C_{opt}) \cap I(\mu_i)] + |C_{opt}|$ according to Lemma 6. Realizing instead of $C_{opt}$ just the set of cycles $C_{M'} = C'_{opt}$ corresponding to $M'$ drops at most $|C_{opt}|/2$ facial cycles in $C$, while imposing weaker interface constraints on the children. We therefore have

$$T[\mu, I] = \sum_{i=1}^{k} T[\mu_i, (I \cup C_{opt}) \cap I(\mu_i)] + |C_{opt}| \leq \sum_{i=1}^{k} T[\mu_i, (I \cup C_{M'}) \cap I(\mu_i)] + 2|M'|$$

We now use the fact that the $\tilde{T}[\mu_i, \cdot]$ are $c$-approximations of the $T[\mu_i, \cdot]$, and hence also $c'$-approximations for $c' = \max\{c, 2\}$, and we also separate the sum by the index set $J_1$ and $J_2$ and consider the two matchings $M'_1$ and $M'_2$ separately.

$$\sum_{i=1}^{k} T[\mu_i, (I \cup C_{M'}) \cap I(\mu_i)] + 2|M'| \leq c' \cdot \sum_{i \in J_1} \tilde{T}[\mu_i, (C_{M'_1} \cup I) \cap I(\mu_i)] + 2|M'_1| + c' \cdot \sum_{i \in J_2} \tilde{T}[\mu_i, (C_{M'_2} \cup I) \cap I(\mu_i)] + 2|M'_2|.$$  \hfill (1)

Observe that the indices of the children intersected by cycles that form a matching $M_2$ in $H$ are all contained in $J_2$. By the definition of $H$, we have $\tilde{T}[\mu_i, (C_{M'_2} \cup I) \cap I(\mu_i)] = \tilde{T}[\mu_i, I \cap I(\mu_i)]$, for $i \in J_2$.

For the first term, observe that, for each edge $\mu_i\mu_j \in M'_1$, we have $\tilde{T}[\mu_x, (M'_1 \cup I) \cap I(\mu_x)] \leq \tilde{T}[\mu_x, I \cap I(\mu_x)] - 1$ for at least one $x \in \{i, j\}$. Otherwise the edge would be in $H$, and hence in $M'_2$. Let $J'_1 \subseteq J_1$ denote the set of indices where this happens and let $J'_2 = J_1 \setminus J'_1$. 


Observe that \( |J_i'| \geq |M'_c| \). We thus have
\[
c' \cdot \sum_{i \in J_i} \tilde{T}[\mu_i, (C_{M'_c} \cup I) \cap \mathcal{I}(\mu_i)] + 2|M'_c| \\
= c' \cdot \sum_{i \in J_i} \tilde{T}[\mu_i, (C_{M'_c} \cup I) \cap \mathcal{I}(\mu_i)] + 2|M'_c| \\
\leq \sum_{i \in J_i} (\tilde{T}[\mu_i, I \cap \mathcal{I}(\mu_i)] - 1) + c' \cdot \sum_{i \in J_i} \tilde{T}[\mu_i, I \cap \mathcal{I}(\mu_i)] + 2|M'_c| \\
\leq c' \cdot \left( \sum_{i \in J_i} \tilde{T}[\mu_i, I \cap \mathcal{I}(\mu_i)] - |J_i'| \right) + 2|M'_c| \leq c' \cdot \sum_{i \in J_i} \tilde{T}[\mu_i, I \cap \mathcal{I}(\mu_i)],
\]
where the last step uses the fact that \( c' \geq 2 \). Plugging this information into Eq. 1, yields
\[
c' \cdot \sum_{i \in J_i} \tilde{T}[\mu_i, (C_{M'_c} \cup I) \cap \mathcal{I}(\mu_i)] + 2|M'_c| + c' \cdot \sum_{i \in J_i} \tilde{T}[\mu_i, (C_{M'_c} \cup I) \cap \mathcal{I}(\mu_i)] + 2|M'_c| \\
\leq c' \cdot \sum_{i=1}^{k} \tilde{T}[\mu_i, I \cap \mathcal{I}(\mu_i)] + 2|M'| \leq c' \cdot \sum_{i=1}^{k} \tilde{T}[\mu_i, I \cap \mathcal{I}(\mu_i)] + 2|M| \\
\leq c' \cdot \left( \sum_{i=1}^{k} \tilde{T}[\mu_i, I \cap \mathcal{I}(\mu_i)] + |M| \right) = c' \cdot \left( \sum_{i=1}^{k} \tilde{T}[\mu_i, (I \cup C_{M}) \cap \mathcal{I}(\mu_i)] + |M| \right).
\]

The last three steps use the facts that \( M \subseteq E(H) \) is a maximum matching, and hence larger than \( M'_2 \), that \( c' \geq 2 \), and that \( C_M \subseteq E(H) \), respectively.

We note that the bottleneck for computing \( T[\mu, I] \) is finding a maximum matching in a graph with \( O(|\text{skel}(\mu)|) \) vertices and \( O(|\mathcal{C}|) \) edges. Hence the running time for one step is \( O(|\text{skel}(\mu)| + \sqrt{|\text{skel}(\mu)|} \cdot |\mathcal{C}|) \). Since \( |I(\mu)| \leq |\mathcal{C}|^2 \), the running time for processing a single P-node \( \mu \) is \( O(|\text{skel}(\mu)| |\mathcal{C}|^2 + \sqrt{|\text{skel}(\mu)|} \cdot |\mathcal{C}|^3) \). The total time for processing all P-nodes is \( O(n|\mathcal{C}|^2 + \sqrt{n}|\mathcal{C}|^3) \).

\[ \textbf{Theorem 16.} \text{There is a 2-approximation algorithm with running time } O(n|\mathcal{C}|^2 + \sqrt{n}|\mathcal{C}|^3) \text{ for MAX FACIAL } \mathcal{C} \text{-CYCLES in series-parallel graphs.} \]

Next we deal with R-nodes. Let \( \mu \) be an R-node with children \( \mu_1, \ldots, \mu_k \). For each face \( f \) of \( \text{skel}(\mu) \) let \( J_f \) denote the indices of the children \( \mu_i \) whose corresponding virtual edge in \( \text{skel}(\mu) \) is incident to \( f \).

Fix \( I \in \mathcal{I}(\mu) \). We propose the following algorithm for computing \( \tilde{T}[\mu, I] \). Consider the subgraph \( H \) of the dual of \( \text{skel}(\mu) \) induced by those vertices \( v \) corresponding to a face \( f \) not incident to the parent edge of \( \text{skel}(\mu) \) and such that there exists a cycle \( C_v \in \mathcal{C} \) that projects to the boundary of \( f \) and such that \( \tilde{T}[\mu_i, (\{C_v \cup I) \cap \mathcal{I}(\mu_i)] = \tilde{T}[\mu_i, I \cap \mathcal{I}(\mu_i)] \), i.e., requiring that \( C_v \) is realized in \( \mu_i \) does not change the approximate number of faces of pert(\( \mu_i \)) in \( C \).

Now we compute a \((1 + \varepsilon/4)\)-approximation of a maximum independent set of \( H \), which can be done in time polynomial in \(|\text{skel}(\mu)| \) (and exponential in \((1/\varepsilon) \)) [3]. Let \( X \) denote this independent set, and let \( C_X = \{ C_v \mid v \in X \} \) be a set of corresponding cycles in \( \mathcal{C} \). We set \( \tilde{T}[\mu, I] = \sum_{v=1}^{k} \tilde{T}[\mu_i, (I \cup X) \cap \mathcal{I}(\mu_i)] + |X| = \sum_{v=1}^{k} \tilde{T}[\mu_i, I \cap \mathcal{I}(\mu_i)] + |X| \), and claim that in this fashion \( \tilde{T}[\mu, I] \) is a \( \max(c, (4 + \varepsilon)/4) \)-approximation provided that \( \tilde{T}[\mu_i, \cdot] \) is a \( c \)-approximation of \( T[\mu_i, \cdot] \). The proof is similar to that of Lemma 15. It 4-colors the facial cycles \( C_{opt} \subseteq \mathcal{C} \) of an optimal solution and considers the largest color class, which is an independent set of faces that has size at least \(|C_{opt}|/4\).
Lemma 17. Let $\tilde{T}[$μ,·$]$ be the table computed in the above fashion. Then, $\tilde{T}[$μ,·$]$ is a $\max\{c,(4+\varepsilon)\}$-approximation of $T[$μ,·$]$ provided that $\tilde{T}[$μ,$i$,...] is a $c$-approximation of $T[$μ,$i$,...$]$. Overall, we obtain the following theorem.

Theorem 18. MAX FACIAL C-CYCLES for biconnected planar graphs admits an efficient $(4 + \varepsilon)$-approximation algorithm for any $\varepsilon > 0$.

6 Conclusions

In this paper, we explored the boundaries of the computational complexity of MAX FACIAL C-CYCLES. In particular, we proved the problem NP-hard under restrictive conditions, showed that slightly stronger conditions make the problem tractable, and gave constant-factor approximations for series-parallel and biconnected planar graphs with approximation guarantees of 2 and $4 + \varepsilon$ for any $\varepsilon > 0$, respectively. Our main open question is whether these approximation guarantees may be improved.

References


An Algorithm for the Maximum Weight Strongly Stable Matching Problem

Adam Kunysz
Institute of Computer Science, University of Wrocław, Poland
aku@cs.uni.wroc.pl

Abstract
An instance of the maximum weight strongly stable matching problem with incomplete lists and ties is an undirected bipartite graph $G = (A \cup B, E)$, with an adjacency list being a linearly ordered list of ties, which are vertices equally good for a given vertex. We are also given a weight function $w$ on the set $E$. An edge $(x, y) \in E \setminus M$ is a blocking edge for $M$ if by getting matched to each other neither of the vertices $x$ and $y$ would become worse off and at least one of them would become better off. A matching is strongly stable if there is no blocking edge with respect to it. The goal is to compute a strongly stable matching of maximum weight with respect to $w$.

We give a polyhedral characterisation of the problem and prove that the strongly stable matching polytope is integral. This result implies that the maximum weight strongly stable matching problem can be solved in polynomial time. Thereby answering an open question by Gusfield and Irving [6]. The main result of this paper is an efficient $O(nm \log(Wn))$ time algorithm for computing a maximum weight strongly stable matching, where we denote $n = |V|$, $m = |E|$ and $W$ is a maximum weight of an edge in $G$. For small edge weights we show that the problem can be solved in $O(nm)$ time. Note that the fastest known algorithm for the unweighted version of the problem has $O(nm)$ runtime [9]. Our algorithm is based on the rotation structure which was constructed for strongly stable matchings in [12].

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases Stable marriage, Strongly stable matching, Weighted matching, Rotation

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.42

Funding Partly supported by Polish National Science Center grant UMO-2013/11/B/ST6/01748.

1 Introduction
An instance of the Stable Marriage Problem with Ties and Incomplete Lists (smti) is an undirected bipartite graph $G = (A \cup B, E)$, with an adjacency list being a linearly ordered list of ties, which are vertices equally good for a given vertex. Ties are disjoint and may contain one vertex. Let $b_1$ and $b_2$ be two vertices incident to $a$ in $G$. Depending on the preference of a one of the following holds. (1) $a$ (strictly) prefers $b_1$ to $b_2$ - denoted as $b_1 >_a b_2$, (2) $a$ is indifferent between $b_1$ and $b_2$ - denoted as $b_1 =_a b_2$, (3) $a$ (strictly) prefers $b_2$ to $b_1$ - denoted as $b_1 <_a b_2$. If $a$ prefers $b_1$ to $b_2$ or is indifferent between them then we say that $a$ weakly prefers $b_1$ to $b_2$ and denote it as $b_1 \geq_a b_2$.

An edge $(a, b) \in E \setminus M$ is a blocking edge with respect to $M$ if by getting matched with each other neither of the vertices $a$ and $b$ would become worse off and at least one of them would become better off than in $M$. Formally an edge $(a, b) \in E \setminus M$ is blocking if either $a >_b M(b)$ and $b \geq_a M(a)$ or $a >_b M(b)$ and $b >_a M(a)$ hold.

By $M(a)$ we denote a partner of $a$ in the matching $M$. If $a$ is unmatched in $M$ we abuse the notation and write $b >_a M(a)$ for each $(a, b) \in E$. We assume that every vertex prefers
to be matched to its neighbour in $G$ rather than to remain unmatched. We say that a matching is strongly stable if there is no blocking edge with respect to it.

We study a version of the problem where besides the graph $G$ and preference lists we are also given a weight function $w : E \rightarrow \mathbb{N}$. We define the weight of a matching $M$ to be $w(M) = \sum_{e \in M} w(e)$. The goal is to find a strongly stable matching $M$ maximising $w(M)$.

**Motivation.** The stable matching problem and its extensions have widespread application to matching schemes [19]. One of the most known examples are the labor market for medical interns and the college admissions market.

It is known that the deferred acceptance algorithm [5] calculates a stable matching optimal for one side of the market. The extension to the weighted variant of the problem allows us to define suitable objective functions and use them to obtain various optimal stable matchings.

The notion of strong stability allows us to prevent the following scenarios. Suppose that agent $a$ is matched to $M(a)$ and $a$ is indifferent between $M(a)$ and $b$. Also assume that $b$ prefers $a$ over $M(b)$. The agent $b$ to improve their situation may be inclined to use an action, like bribery, to convince $a$ to accept them. Since $a$ would not get worse and $b$ would get better by getting matched to each other, they might undermine the current assignment.

**Previous results.** The variant of the problem with strict preferences known as the stable marriage problem (SMP) has been extensively studied in the literature. In their seminal paper Gale and Shapley [5] showed that every instance of the problem admits a stable matching and described an $O(n + m)$ time algorithm for computing such a matching. Many structural properties of the problem have been described over the years. In [6] Gusfield and Irving have proven that the set of stable marriage solutions forms a distributive lattice. They also show that even though the lattice can be of exponential size, it can be compactly represented as a set of closed sets of a certain partial order on $O(m)$ elements. The representation can be built in $O(m)$ time based on the notion of rotation.

Vande Vate [25] initiated the study of the stable marriage problem using the polyhedral approach. He described a stable marriage polytope and showed its integrality. His description has been extended by Rothblum [21] to the case of incomplete preference lists. In subsequent papers several simpler proofs of the integrality of the stable marriage polytope have been given [20], [24]. It has been also proven that any fractional solution in the stable marriage polytope can be expressed as a convex combination of integral solutions [24]. These results imply that the maximum weight stable marriage problem can be solved in polynomial time.

Several efficient algorithms for this problem have been developed over the years. Gusfield and Irving [6] described an $O(m^2 \log n)$ algorithm. The authors exploit the rotation structure and reduce the problem to finding a maximum weight closed subset of a poset. This classical problem can in turn be reduced to computing a maximum flow. The flow network obtained from the reduction consists of $O(m)$ nodes and $O(m)$ edges. Gusfield and Irving use $O(nm \log n)$ algorithm by Sleator and Tarjan [23] to solve the maximum flow problem and obtain $O(m^2 \log n)$ complexity. A faster maximum flow algorithm would lead to the improvement in their algorithm. Feder [2] showed that if $K = O((m/\log^2 m)^2)$ then the weighted stable marriage problem can be solved in $O(m\sqrt{K})$ time and $O(nm \log K)$ for arbitrary $K$ where $K$ is the weight of the solution. Note that algorithms by Gusfield and Irving and by Feder assume a certain monotonicity condition on edge weights, however in the case of bipartite graphs this condition can be dropped as we show later.
The problem of computing a strongly stable matching in instances of smti has received a significant attention in the literature. Irving [7] gave an $O(n^3)$ algorithm for the problem under the assumption that the graph is complete and there is an equal number of men and women. In [14] Manlove extended this algorithm to incomplete bipartite graphs. His algorithm has $O(m^2)$ time complexity. Kavitha et al. [9] gave an $O(nm)$ algorithm for the problem. The structure of the set of solutions to the problem has been proven to be similar to the structure of the case of no ties. In [15] Manlove has proven that the set of solutions forms a distributive lattice. Recently, Kunysz et al. [12] gave an $O(nm)$ algorithm for constructing a compact representation of the lattice and generalized the notion of rotation to the strong stability setting. To the best of our knowledge the weighted version of the strongly stable matching problem has not been studied in the literature yet.

Our results. Gusfield and Irving [6] asked whether there is an LP representation of an instance of smti under strong stability similar to the case of no ties. The problem was again posed by Manlove [16] in his recent book. We solve this problem, adapting techniques used in [24] to our setting. We prove that any fractional solution to the polytope can be expressed as a convex combination of integral solutions. Thus the polytope is integral and the maximum weight strongly stable matching problem can be solved in polynomial time.

A natural question is whether the rotation structure can be exploited to obtain a faster algorithm. We answer this question affirmatively and give an $O(nm \log(Wn))$ algorithm, where $W$ is the maximum weight of an edge. We also show that if $W$ is sufficiently small then the problem can be solved in $O(nm)$ time. The technique of Gusfield and Irving cannot be directly applied to our problem. In the setting without ties the authors base their algorithm on the fact that there is a one-to-one correspondence between stable matchings and closed sets of a certain poset of size $O(m)$. In our problem a similar one-to-one correspondence exists between equivalence classes of strongly stable matchings under a certain equivalence relation and closed sets of a poset of size $O(m)$. The correspondence allows us to represent exactly one matching from each equivalence class based on a computation of so called maximal sequence of strongly stable matchings. The main obstacle is that each equivalence class may contain exponentially many matchings and there is a possibility that a represented matching is not of maximum weight within its class. The primary novelty of this paper is an algorithm for computing so called heavy maximal sequence of strongly stable matchings, which allows us to represent a matching of maximum weight from each equivalence class. As a result we reduce our problem to finding a maximum weight closed set of a poset, and solve this problem using Feder algorithm [2].

Related work. Stable matchings have been extensively studied in non-bipartite instances with strict preferences. Feder [1] has shown that in this setting the maximum weight stable matching problem is $NP$-hard and he gave a 2-approximation algorithm for the problem.

In smti instances three different notions of stability can be defined depending on the definition of a blocking edge. Namely weak, strong and super stability. Weakly stable matchings can be of different sizes. Iwama et al. [8] have proven that the problem of finding a maximum size weakly stable matching is $NP$-hard. Several approximation algorithms are known for the problem [17], [10], [18]. It is also known that the weighted version of the problem is $NP$-hard and it is not approximable within a factor $n^{1-\epsilon}$ for any $\epsilon > 0$ unless $P = NP$ [13]. The structure of stable matchings under the notion of super stability is well understood. In [3] Fleiner et al. gave a reduction to the 2-SAT problem which results in fast algorithms for a range of problems related to finding “optimal” super stable matchings.
An Algorithm for the Maximum Weight Strongly Stable Matching Problem

2 Preliminaries

Let $\mathcal{I}$ denote an instance of SMTI. Denote the set of all strongly stable matchings in $\mathcal{I}$ by $\mathcal{M}(\mathcal{I})$. Let $V(\mathcal{I})$ and $E(\mathcal{I})$ be respectively sets of vertices and edges of the underlying bipartite graph $G = (A \cup B, E(\mathcal{I}))$ of $\mathcal{I}$. As is customary we call the vertices of $A$ and $B$ respectively men and women. We say that an instance $\mathcal{I}$ is solvable if there is a strongly stable matching in $G$. We define the rank of $w$ in $v$’s preference list, denoted by $\text{rank}(v, w)$, to be 1 plus the number of ties which are preferred to $w$ by $v$. A matching is $\text{man-optimal}$ if every man gets the best partner among all his possible partners in any strongly stable matching.

Theorem 1 ([9]). There is an $O(mn)$ algorithm to determine a man-optimal strongly stable matching of the given instance or report that no strongly stable matching exists.

2.1 Lattice Structure

In this subsection we give a brief overview of results related to the lattice structure of $\mathcal{M}(\mathcal{I})$. As we will see later the lattice can be of exponential size, however its representation of polynomial size can be constructed. Such a representation is described in the next subsection.

Theorem 2 (Rural Hospitals Theorem, [14]). In a given instance of SMTI, the same vertices are matched in all strongly stable matchings.

We define an equivalence relation $\sim$ on $\mathcal{M}(\mathcal{I})$ as follows. For two strongly stable matchings $M$ and $N$, $M \sim N$ if and only if each man $m$ is indifferent between $M(m)$ and $N(m)$. Denote by $[M]$ the equivalence class containing $M$ and denote by $\mathcal{X}$ the set of equivalence classes of $\mathcal{M}(\mathcal{I})$ under $\sim$.

Strongly stable matchings belonging to the same equivalence class can be easily characterised. For a given strongly stable matching $M$ we define an auxiliary graph $H_M = (V', E')$ where $V'$ is the set of vertices matched in $M$ and $E' = \{(a, b) \in E : a, b \in V' \land b =_a M(a) \land a =_b M(b)\}$. The following lemma characterises the set $[M]$.

Lemma 3 ([15]). Let $M \in \mathcal{M}(\mathcal{I})$. Then $M'$ is a strongly stable matching such that $M' \sim M$ if and only if $M'$ is a perfect matching in $H_M$.

For two strongly stable matchings $M$ and $N$ we say that $M$ dominates $N$ and write $N \preceq M$ if each man $m$ weakly prefers $M(m)$ to $N(m)$. If $M$ dominates $N$ and there exists a man $m$ who strictly prefers $M(m)$ to $N(m)$ then we say that $M$ strictly dominates $N$, denote it by $N < M$ and we call $N$ a successor of $M$. Next we define a partial order $\preceq^*$ on $\mathcal{X}$. For any two equivalence classes $[M]$ and $[N]$, we define $[M] \preceq^* [N]$ if and only if $M \preceq N$.

Let $M$ and $N$ be two strongly stable matchings. Consider the symmetric difference $M \oplus N$. Theorem 2 implies that this set contains only alternating cycles.

Lemma 4 ([15]). Let $M$ and $N$ be two strongly stable matchings. Consider any alternating cycle $C$ of $M \oplus N$. Let $(m_0, w_0, m_1, w_1, \ldots, m_{k-1}, w_{k-1})$ be the sequence of vertices of $C$ where $m_i$ are men and $w_i$ are women. Then there are only three possibilities:

- $(\forall m_i)w_i =_{m_i} w_{i+1}$ and $(\forall w_i)m_i =_{w_i} m_{i-1}$
- $(\forall m_i)w_i <_{m_i} w_{i+1}$ and $(\forall w_i)m_i >_{w_i} m_{i-1}$
- $(\forall m_i)w_i >_{m_i} w_{i+1}$ and $(\forall w_i)m_i <_{w_i} m_{i-1}$

Subscripts are taken modulo $k$. 

Below we introduce two operations transforming pairs of strongly stable matchings into other strongly stable matchings. Let $M$ and $N$ be two strongly stable matchings. Consider any man $m$ and his partners $M(m)$ and $N(m)$. By $M \land N$ we denote the matching such that if $M(m) \succeq m N(m)$ then $(m, M(m)) \in M \land N$ and if $M(m) \prec_m N(m)$ then $(m, N(m)) \in M \land N$. Similarly by $M \lor N$ we denote the matching such that if $M(m) \succ_m N(m)$ then $(m, N(m)) \in M \lor N$ and if $M(m) \preceq_m N(m)$ then $(m, M(m)) \in M \lor N$.

It is proven in [15] that both $M \lor N$ and $M \land N$ are strongly stable matchings, and $M, N \preceq M \lor N$ and $M, N \succeq M \land N$. Operations $\lor$ and $\land$ can be extended to the set $\mathcal{X}$. For $[M], [N] \in \mathcal{X}$ we simply define $[M] \lor [N] = [M \lor N]$, $[M] \land [N] = [M \land N]$.

A lattice is a partially ordered set in which every two elements $a, b$ have a unique infimum (denoted $a \lor b$) and a unique supremum (denoted $a \land b$). A lattice $L$ with operations $\lor$ and $\land$ is distributive if for any three elements $x, y, z$ of $L$ the following holds: $x \land (y \lor z) = (x \land y) \lor (x \land z)$.

\textbf{Theorem 5} ([15]). The partial order $(\mathcal{X}, \preceq)$ with operations $\lor$ and $\land$ defined above forms a distributive lattice.

### 2.2 Rotations

In this subsection we review known theorems about rotations in instances of smti under strong stability. These results were previously given in [12] and [11].

Let $M$ and $N$ be two strongly stable matchings such that $N \prec M$. We say that $N$ is a strict successor of $M$ if and only if there is no strongly stable matching $M'$ such that $N \prec M' \prec M$. Let $M_0$ be a man-optimal strongly stable matching, and let $M_2$ be a woman optimal strongly stable matching. We call a sequence $(M_0, M_1, \ldots, M_z)$ such that $M_0 \succ M_1 \succ \ldots \succ M_z$ and $M_{i+1}$ is a strict successor of $M_i$, a maximal sequence of strongly stable matchings.

\textbf{Theorem 6} ([12]). There is an $O(nm)$ time algorithm to compute a maximal sequence of strongly stable matchings.

Let $M$ and $N$ be two strongly stable matchings such that $N$ is a strict successor of $M$. Consider some matchings $M' \in [M]$, $N' \in [N]$. Note that from the definition of $\sim$ it follows that for every vertex $v$ we have $rank(v, M(v)) + rank(v, N(v)) = rank(v, M'(v))$ and $rank(v, N'(v))$. In other words when we transform a matching from $[M]$ into some matching from $[N]$, the change of $v$’s rank does not depend on the choice of matchings from equivalence classes. This observation motivates the definition of rotation.

Let $M$ and $N$ be two strongly stable matchings such that $N$ is a strict successor of $M$. For any vertex $v$ denote $r_v = rank(v, M(v))$ and $r'_v = rank(v, N(v))$. We say that a set of triples $\rho([M], [N]) = \{(v, r_v, r'_v) : v \in V(I), r_v \neq r'_v\}$ is a rotation transforming $[M]$ into $[N]$.

Let $\rho$ be a rotation and $M, N$ be two strongly stable matchings such that $N$ is a strict successor of $M$. We say that the set of alternating cycles $M \ominus N$ realizes a rotation $\rho$ if $\rho = \rho([M], [N])$. There are potentially many sets of cycles realizing a given rotation. A rotation $\rho$ is exposed in $[M]$ if $\rho = \rho([M], [N])$ for some $N$ which is a strict successor of $M$. We say that $\rho = \rho([M], [N])$ transforms $M'$ into $N'$ if $M' \in [M]$ and $N' \in [N]$.

\textbf{Theorem 7} ([12]). Let $S = (M_0, M_1, \ldots, M_z)$ be a maximal sequence of strongly stable matchings. For $i \in \{0, 1, \ldots, z - 1\}$ denote $\rho_i = \rho([M_i], [M_{i+1}])$. Then the set $D(I) = \{\rho_0, \rho_1, \ldots, \rho_{z-1}\}$ does not depend on the choice of $S$, and $\rho_i \neq \rho_j$ for $i \neq j$. 

ISAAC 2018
Note that given a maximal sequence of strongly stable matchings \( S = (M_0, M_1, \ldots, M_z) \) we can easily compute rotations \((\rho_0, \rho_1, \ldots, \rho_{z-1})\) where \( \rho_i = \rho([M_i], [M_{i+1}]) \). Moreover the set \( C_S(\rho_i) = M_i \oplus M_{i+1} \) realizes \( \rho_i \) for each \( i \).

**Definition 8.** Let \( D(I) \) be the set of all rotations in \( I \). We define the order \( \prec \) on elements of \( D(I) \) as follows. We say that a rotation \( \rho \) precedes rotation \( \rho' \) and write \( \rho \prec \rho' \) if and only if for every maximal sequence \( S = (M_0, M_1, \ldots, M_z) \) of strongly stable matchings we have \( \rho = \rho([M_i], [M_{i+1}]) \) and \( \rho' = \rho([M_j], [M_{j+1}]) \) for some \( i, j \) such that \( i < j \).

Let \( Z \) be a subset of \( D(I) \). We say that \( Z \) is a closed set if there is no \( \rho \in D(I) \setminus Z \) such that \( \rho \prec \rho' \) for some \( \rho' \in Z \). It turns out that each closed set corresponds to an equivalence class of \( \sim \). Given \( Z \) we can efficiently find an equivalence class corresponding to it.

Assume that we are given a maximal sequence \( S = (M_0, M_1, \ldots, M_z) \) of strongly stable matchings, the set of rotations \( D(I) \), and for each rotation \( \rho_i = \rho([M_i], [M_{i+1}]) \) a set of cycles \( C_S(\rho_i) = M_i \oplus M_{i+1} \) realizing it. Let \( Z = \{\rho_{a_0}, \rho_{a_1}, \ldots, \rho_{a_{k-1}}\} \) be a closed set. We order its elements so that there are no \( i, j \) such that \( i < j \) and \( \rho_{a_i} \succ \rho_{a_j} \). We define a sequence of strongly stable matchings \( N_0 = M_0, N_{i+1} = N_i \oplus C_S(\rho_{a_i}) \). We denote \( f_S(Z) = N_k \). Note that the sequence \( \{N_i\} \) depends on the ordering of elements of \( Z \), however its last element \( f_S(Z) = M_0 \oplus C_S(\rho_{a_0}) \oplus C_S(\rho_{a_1}) \oplus \ldots \oplus C_S(\rho_{a_{k-1}}) \) is the same regardless of the ordering.

**Lemma 9.** For each equivalence class \( [M] \) there is a closed set \( X \) such that \( f_S(X) \in [M] \). Let \( Z_1 \) and \( Z_2 \) be closed sets. Then \( Z_1 \neq Z_2 \) implies that \( [f_S(Z_1)] \neq [f_S(Z_2)] \).

For each closed set \( Z \) we define \( g_S(Z) = [f_S(Z)] \). It can be proven that \( g_S \) does not depend on the choice of \( S \) and that \( g_S \) is a bijection between closed sets of \( D(I), \prec \) and the set \( X \). The above discussion is summarized in the following theorem.

**Theorem 10 ([12]).** There is a one-to-one correspondence between the set \( X \) of equivalence classes of \( \sim \) and the closed sets of \( D(I), \prec \).

It is important to note that given the function \( f_S \) we can get one strongly stable matching from each equivalence class and that depending on the choice of \( S \) these matchings may differ. In other words if \( S \neq S' \) then it may happen that \( f_S(Z) \neq f_{S'}(Z) \) for some \( Z \), however regardless of the choice of \( S \) and \( S' \) we have \([f_S(Z)] = [f_{S'}(Z)] \).

Note that from Definition 8 alone it is non-trivial how to efficiently construct the relation \( \prec \) on \( D(I) \). Construction of an explicit representation of the relation \( \prec \) would take \( \Omega(m^2) \) time, because \( D(I) \) might have \( \Omega(m) \) elements.

**Theorem 11 ([12]).** There is a graph \( G' = (D(I), E') \) such that \( |E'| = O(m) \), and the closed sets in \( G' \) are exactly the same as the closed sets in the poset \( (D(I), \prec) \). Such a graph can be constructed in \( O(mn) \) time.

### 3 Strongly Stable Matching Polytope

Let us denote the set of men as \( A = \{a_1, a_2, \ldots, a_p\} \) and the set of women as \( B = \{b_1, b_2, \ldots, b_q\} \). Additionally by \( P_{SSM} \) we denote a strongly stable matching polytope described by the following set of inequalities.
\[
\sum_{j=1}^{q} x_{i,j} \leq 1 \quad \forall i(1 \leq i \leq p) \quad (1)
\]

\[
\sum_{i=1}^{p} x_{i,j} \leq 1 \quad \forall j(1 \leq j \leq q) \quad (2)
\]

\[
x_{i,j} \geq 0 \quad \forall (i,j)(1 \leq i \leq p, 1 \leq j \leq q) \quad (3)
\]

\[
\sum_{k:b_k = a_i b_j} x_{i,k} + \sum_{k:a_k = b_j} x_{k,j} + \sum_{k:a_k = a_i} x_{i,k} \geq 1 \quad \forall (i,j) \quad (4)
\]

\[
\sum_{k:b_k = a_i b_j} x_{i,k} + \sum_{k:a_k = b_j} x_{k,j} + \sum_{k:a_k = a_i} x_{i,k} \geq 1 \quad \forall (i,j) \quad (5)
\]

Inequalities (1), (2) and (3) are standard matching constraints. If \( x \in P_{SSM} \) is an integral solution, then constraints (4) and (5) for an edge \((a_i, b_j)\) imply that \((a_i, b_j)\) does not block the matching associated with \( x \). Thus integral solutions of \( P_{SSM} \) are exactly strongly stable matchings of \( G \). We call such solutions strongly stable matching solutions.

Note that if there are no ties in the instance then the terms \( \sum_{k:a_k = b_j} x_{i,k} \) and \( \sum_{k:b_k = a_i b_j} x_{i,k} \) in (4) and (5) reduce to \( x_{i,j} \) and the description of the polytope is identical to the well known description of the stable marriage polytope (see [24]). The proof of the next lemma is based on self-duality of the associated linear program and complementary slackness conditions.

**Lemma 12.** Let \( x \in P_{SSM} \) be a feasible solution. Then for each \( 1 \leq i \leq p, 1 \leq j \leq q \) the following hold:

\[
x_{i,j} > 0 \Rightarrow \sum_{k:b_k = a_i b_j} x_{i,k} + \sum_{k:a_k = b_j} x_{k,j} + \sum_{k:a_k = a_i} x_{i,k} = 1
\]

\[
x_{i,j} > 0 \Rightarrow \sum_{k:b_k = a_i b_j} x_{i,k} + \sum_{k:a_k = b_j} x_{k,j} + \sum_{k:a_k = a_i} x_{i,k} = 1
\]

\[
x_{i,j} > 0 \Rightarrow \sum_{k=1}^{q} x_{i,k} = 1
\]

\[
x_{i,j} > 0 \Rightarrow \sum_{k=1}^{p} x_{k,j} = 1
\]

It is important to note that for each feasible solution \( x \) if \( x_{i,j} > 0 \) then \( \sum_{k:a_k = b_j} x_{k,j} = \sum_{k:b_k = a_i b_j} x_{i,k} \). Lemma 12 allows us to prove Theorem 13 which shows that each fractional solution to \( P_{SSM} \) can be expressed as a convex combination of strongly stable matchings. The proof is constructive and given a fractional solution one can obtain matchings constituting such a convex combination. Theorem 13 also implies that \( P_{SSM} \) is integral.

**Theorem 13.** The polytope \( P_{SSM} \) is the convex hull of the strongly stable matching solutions.

**Proof.** Let \( x \in P_{SSM} \) be a feasible solution. For each man \( a_i \) such that \( x_{i,j} > 0 \) for some \( j \) we perform the following construction. From Lemma 12 it follows that \( \sum_{k=1}^{q} x_{i,k} = 1 \). For \( a_i \) we arrange all the \( x_{i,k} \) for \( k = 1, 2, \ldots, q \) in order of decreasing preference for \( a_i \). If there are any ties we pick an arbitrary order amongst tied variables. We cover the interval
(0, 1] with smaller intervals \((v_{i,k}, v_{i,k} + x_{i,k}]\) where intervals are arranged in the same order as variables \(x_{i,k}\). We slightly abuse the notation here and by \(x_{i,k}\) we denote corresponding interval \([v, v + x_{i,k}]\). We denote such an arrangement as \(X_i\). Similarly for each woman \(b_j\) such that \(x_{i,j} > 0\) for some \(i\) we construct an arrangement \(Y_j\). The difference is that for women we order intervals in the increasing order of preference and we again order tied variables arbitrarily. Let us by \(T_j(i)\) denote the interval spanned by all intervals \(x_{i,k}\) such that \(b_k = a_i b_j\). Note that such intervals are next to each other in the arrangement. Similarly by \(T'_j(i)\) we denote the interval spanned by \(x_{k,j}\) such that \(a_k = b_j a_i\).

Let \(u\) be any real number belonging to \((0, 1]\). We first construct an auxiliary graph \(H_u = (A' \cup B', F)\) as follows. Let \(A' \subseteq A\) and \(B' \subseteq B\) be sets of men and women for which we created arrangements \(X_i, Y_j\), i.e., \(A' = \{a_i : 1 \leq i \leq p \land \exists j(x_{i,j} > 0)\}\) and \(B' = \{b_j : 1 \leq j \leq q \land \exists i(x_{i,j} > 0)\}\). For each man \(a_i\) if \(u\) lies in the subinterval spanned by \(x_{i,j}\), we add to \(F\) edges corresponding to variables in the tie \(T_i(j)\) in \(X_i\). Obviously each man is indifferent between all the edges incident to him. We now prove that this holds for women as well. Note that from Lemma 12 it follows that if \(x_{i,j} > 0\) then intervals \(T_i(j)\) and \(T'_j(i)\) coincide in arrangements \(X_i\) and \(Y_j\). Let us assume that there are two edges \((a_i, b_j), (a_k, b_j)\) in \(H_u\). Then \(u\) lies in the subintervals spanned by \(T_i(j)\) and \(T_k(j)\). So in particular \(u\) lies in the subintervals spanned by \(T'_j(i)\) and \(T'_j(k)\). This implies that \(T'_j(i)\) and \(T'_j(k)\) are identical so we have \(a_i = b_j a_k\). Hence each woman is indifferent between edges incident to her in \(H_u\).

We are going to show that there exists a perfect matching in \(H_u\). Let us first create a variable \(y\). For each \(i \in A'\) we consider \(X_i\), and assume that \(u\) lies in the subinterval spanned by \(x_{i,j}\). For each \(k\) such that \(x_{i,k} > 0\) and \(b_k = a_i b_j\) we set \(y_{i,k} = \frac{x_{i,k}}{|T_i(j)|}\), where \(|T_i(j)|\) is the length of \(T_i(j)\). From the definition we know that for each \(i\) we have \(\sum_j y_{i,j} = 1\) and similarly for each \(j\) we have \(\sum_i y_{i,j} = 1\). Thus \(y\) is a fractional perfect matching in \(H_u\) and there exists a perfect matching \(M_u\) in \(H_u\) (see [22] for the details of the construction).

We now show that \(M_u\) is strongly stable. Let \(a_i \in A\) be a man matched in \(M_u\) to some \(b_j\). Assume that \(b_k \succ a_i b_j\). In \(X_i\) the tie corresponding to \(x_{i,k}\) lies to the left of the tie corresponding to \(x_{i,j}\). Recall that the tie corresponding to \(x_{i,k}\) coincides in \(X_i\) and \(Y_k\), thus from the construction of \(Y_k\) it follows that \(b_k\) strictly prefers \(M_u(b_k)\) to \(a_i\), hence \((a_i, b_k)\) does not block the matching. We can analogously prove that if there exists \(a_k\) such that \(a_k \succ b_j a_i\) then \((a_k, b_j)\) does not block the matching. Thus \(M_u\) is strongly stable.

It remains to show how to express \(x\) as a convex combination of strongly stable matchings. Note that as we move \(u\) from 0 to 1 graphs \(H_u\) change. We denote a sequence of graphs that we can obtain in this way by \(H_1, H_2, \ldots, H_q\) and let \((I_1, I_{i+1})\) be an interval corresponding to \(H_i\) for each \(i\). From the discussion above we know that each of the graphs \(H_i\) admits a perfect matching \(M_i\). Let \(y_i\) be the incidence vector of \(M_i\). One can easily see that \(x = \sum_{i=1}^{q-1} (I_{i+1} - I_i) y_i\), thus the theorem holds.

4 Maximum Weight Strongly Stable Matching

In this section we give an efficient algorithm for computing a maximum weight strongly stable matching. We first show that given a matching \(M\) we can easily find a maximum weight matching amongst the ones belonging to \([M]\).

**Definition 14.** We say that a strongly stable matching \(M\) is heavy if for each strongly stable matching \(M'\) such that \(M' \in [M]\) we have \(w(M) \geq w(M')\).

In order to characterise heavy matchings belonging to \([M]\) we first extend the definition of \(H_M\) (see Section 2) so that each edge is of the same weight as in \(G\). The following lemma is a direct consequence of Lemma 3 and allows us to find a heavy matching belonging to a given equivalence class.
Lemma 15. Let $M \in \mathcal{M}(I)$. Then $M'$ is a heavy strongly stable matching such that $M' \sim M$ if and only if $M'$ is a maximum weight perfect matching in $H_M$.

In order to solve the general problem we need the following definition.

Definition 16. Let $S = (M_0, M_1, \ldots, M_z)$ be a maximal sequence of strongly stable matchings. We say that a sequence $S$ is a heavy maximal sequence of strongly stable matchings if $M_i$ is heavy for each $0 \leq i \leq z$.

It turns out that once a heavy maximal sequence of strongly stable matchings is computed, we are able to efficiently find a heavy matching in each equivalence class.

Theorem 17. Let $S = (M_0, M_1, \ldots, M_z)$ be a heavy maximal sequence of strongly stable matchings of $I$. Then for each closed subset of rotations $X \subseteq D(I)$ the matching $f_S(X)$ is heavy.

Before we prove Theorem 17 we need to introduce a few more definitions.

Let $M$ and $N$ be two strongly stable matchings such that $N$ is a strict successor of $M$. We denote by $\rho = \rho([M],[N])$ a rotation transforming $[M]$ into $[N]$ and by $V_\rho = \{v : \exists(a,b) \in \rho\}$ we denote the set of all vertices that change their rank when $\rho$ is applied. Now we define two auxiliary graphs $K_\rho = (V_\rho, E_\rho)$ and $L_\rho = (V_\rho, F_\rho)$. The intuition behind these two graphs is as follows. The graph $L_\rho$ contains all the edges of the original graph that have both endpoints in $V_\rho$ and can potentially belong to matchings from $[M]$. The graph $K_\rho$ fulfills a similar role for the class $[N]$. The set $E_\rho$ is defined as $E_\rho = \{(a,b) \in E(I) : \exists(c,d)((a,c,rank(a,b)) \in \rho \land (b,d,rank(b,a)) \in \rho)\}$. Similarly we define $F_\rho = \{(a,b) \in E(I) : \exists(c,d)((a,rank(a,b),c) \in \rho \land (b,rank(b,a),d) \in \rho)\}$.

Lemma 18. Let $M$, $N$ be two strongly stable matchings such that $N$ is a strict successor of $M$. Assume that $M$ is a heavy matching and $\rho = \rho([M],[N])$ is a rotation transforming $[M]$ into $[N]$. Additionally let $X \in [N]$.

Then $X$ is a heavy matching if and only if the following hold:
1. Edges of the set $X \cap E_\rho$ form a maximum weight perfect matching of $K_\rho$.
2. $w(\{(a,b) \in M : a,b \notin V_\rho\}) = w(\{(a,b) \in X : a,b \notin V_\rho\})$.

Note that given a heavy matching $M$ we can obtain a heavy matching $N' \in [N]$. In order to do so we first compute a maximum weight perfect matching $X$ in $K_\rho$ and then simply take $N' = M \cup X \setminus (M \cap (V_\rho \times V_\rho))$. The above lemma implies that $N'$ is heavy. We are now ready to present the proof of Theorem 17.

Proof of Theorem 17. Let us assume by contradiction that there is a subset $Y \in D(I)$ of rotations such that $f_S(Y)$ is not heavy. Let $Y = \{\rho_1, \rho_2, \ldots, \rho_k\}$. We can assume without the loss of generality that rotations of $Y$ are ordered so that there are no $i,j$ such that $i < j$ and $\rho_i \succ \rho_j$.

We first define a sequence $N_0, N_1, \ldots, N_k$ of strongly stable matchings. Let $N_0 = M_0$ and $N_i = N_{i-1} \sqcup C_S(\rho_i)$ for $0 < i \leq k$. From the initial assumptions we know that $N_k = f_S(Y)$. Moreover we can assume without the loss of generality that $N_k$ is the first matching in the sequence $N_0, N_1, \ldots, N_k$ which is not heavy. Let us denote $\rho' = \rho([N_{k-1}],[N_k])$. From the definition of $S$ we know that there exists $j$ such that $\rho([M_{j-1}],[M_j]) = \rho'$.

From Lemma 18 we know that $M_j \cap E_{\rho'}$ is a maximum weight perfect matching in $K_{\rho'}$. Additionally since $N_{k-1}$ is a heavy matching and $N_k$ is not a heavy matching, we know that at least one of conditions (1) and (2) of Lemma 18 does not hold for $N_{k-1}$ and $N_k$. We are going to prove that (2) holds for $N_{k-1}$ and $N_k$, i.e., $w(\{(a,b) \in N_{k-1} : a,b \notin V_\rho\}) = \ldots$
An Algorithm for the Maximum Weight Strongly Stable Matching Problem

Let \( C \) be any cycle belonging to \( C_S(\rho') \) such that \( C \cap (V_{\rho} \times V_{\rho'}) = \emptyset \) (Note Lemma 4 and the definition of \( \rho' \) imply that each cycle of \( C_S(\rho') \) is either contained in \( V_{\rho} \) or disjoint with this set). Each vertex of \( C \) is indifferent between edges of \( C \) since this cycle does not belong to \( \rho' \). The cycle \( C \) can be partitioned into two matchings \( C \cap M_{j-1} \) and \( C \cap M_j \). One can easily see that we have \( w(C \cap M_{j-1}) = w(C \cap M_j) \) as otherwise either \( w(M_{j-1} \cap C) > w(M_j) \) would hold and this would contradict the assumption that \( M_{j-1} \) and \( M_j \) are both heavy matchings. This implies that \( w(N_{k-1} \cap C) \) and the weight of the matching does not change when cycles of \( C_S(\rho') \) which do not belong to the rotation are applied, thus \( w((a,b) \in N_{k-1} : a,b \notin V_{\rho}) = w((a,b) \in N_k : a,b \notin V_{\rho}) \) holds.

From Lemma 18 it follows that \( N_k \cap E_{\rho'} \) is not a maximum weight perfect matching in \( K_\rho \). Thus we have \( w(M_j \cap E_{\rho'}) > w(N_k \cap E_{\rho'}) \).

Let \( C \) be any cycle of \( C_S(\rho') \) belonging to the rotation \( \rho' \). We will prove that \( C \cap N_k = C \cap M_j \). To see this consider any man \( m \) belonging to \( C \). Exactly two edges \((m, w_1), (m, w_2)\) of \( N_{k-1} \cap N_k \) are incident to \( m \). Since \( m \in V_{\rho} \) we can assume without the loss of generality that \( w_1 \succ m, w_2 \). From \( N_{k-1} \succ N_k \) it follows that \((m, w_2) \in N_k \). We can similarly prove that \((m, w_2) \in M_j \). This implies that \( C \cap N_k = C \cap M_j \) holds. Hence we also have \( M_j \cap E_{\rho'} = N_k \cap E_{\rho'} \) - a contradiction with the fact that \( w(M_j \cap E_{\rho'}) > w(N_k \cap E_{\rho'}) \).

From the above discussion it follows that the lemma holds.

Below we explain how a heavy maximum sequence of strongly stable matchings can be exploited to solve the maximum weight strongly stable matching problem. It turns out that given such a sequence, our problem can be reduced to computing a maximum weight closed subset of a poset, similarly as in the case of no ties.

Let us consider the poset of rotations \( D(\mathcal{I}, \prec) \). We are going to assign a weight to each element of \( D(\mathcal{I}) \). Let \( S' = (M'_0, M'_1, \ldots, M'_r) \) be a heavy maximal sequence of strongly stable matchings. Assume that \( \rho' \in D(\mathcal{I}) \) is a rotation such that \( \rho' = \rho([M'_{j-1}], [M'_j]) \). Let us denote \( w_{S'}(\rho') = w(M'_j) - w(M'_{j-1}) \). We first show that the weight of a rotation does not depend on the choice of a maximal heavy sequence of strongly stable matchings.

**Lemma 19.** Let \( S_1, S_2 \) be two heavy maximal sequences of strongly stable matchings and let \( \rho \in D(\mathcal{I}) \) be a rotation. Then \( w_{S_1}(\rho) = w_{S_2}(\rho) \).

From now on we are going to skip the subscript in the definition of \( w \), i.e., we write \( w(\rho) \) instead of \( w_{S'}(\rho) \). We slightly abuse the notation here, but it should not cause any confusion. From Theorem 10 each closed subset of rotations \( X \subseteq D(\mathcal{I}) \) corresponds to a certain equivalence class \([M]\) of \( \sim \). It turns out that given weights of rotations belonging to \( X \) we can determine the weight of a heavy strongly stable matching belonging to \([M]\).

**Lemma 20.** Assume that \( M \) is a heavy matching and that \( M_0 \) is a heavy man optimal matching. Let \( X_M \subseteq D(\mathcal{I}) \) be a subset of rotations corresponding to \([M]\). Then \( w(M) = w(M_0) + \sum_{\rho \in X_M} w(\rho) \).

The following theorem is a direct consequence of the above lemma.

**Theorem 21.** Let \( M \) be a heavy matching and let \( X_M \subseteq D(\mathcal{I}) \) be a subset of rotations corresponding to \( M \). Then \( M \) is a maximum weight matching of \( \mathcal{I} \) if and only if \( X_M \) is a maximum weight closed subset of \( D(\mathcal{I}, \prec) \) with respect to the weight function \( w \).

A maximum weight closed subset of a poset is a classical problem. In [6] Gusfield and Irving show a reduction to the minimum s-t cut in a graph with \( O(m) \) vertices and edges.
**Algorithm 1** For computing a heavy maximal sequence of strongly stable matchings.

**Input:** $I$ - a solvable instance of smti

1. compute a maximal sequence of strongly stable matchings $S = (M_0, M_1, \ldots, M_z)$
2. compute a heavy matching $M'_0 \in [M_0]
3. for $i = 1, 2, \ldots, z$ do
4. let $\rho_i = \rho([M_{i-1}], [M_i])$
5. compute a maximum weight perfect matching $Y$ in $K_{\rho_i}$
6. let $M'_i = M'_{i-1} \cup Y \setminus (M'_{i-1} \cap (V_{\rho_i} \times V_{\rho_i}))$
7. return $(M'_0, M'_1, \ldots, M'_z)$

This problem can be solved with a standard maximum flow computation, however in the special case of posets obtained from instances of smti we can construct the minimum cut in $O(mn \log(Wn))$ time or in $O(nm)$ time if $W = O(\min\{n, \frac{m}{\log{m}}\})$.

To achieve these complexity bounds we use algorithms of Feder [2]. The author shows that a maximum flow in an uncapacitated network with $m$ edges and of explicit width $q$ can be found in $O(qm \log(K))$ time. It can be shown that in our case we have $q \leq n$ and $\log(K) \leq \log(Wn)$, thus the runtime is $O(nm \log(Wn))$. Feder also shows that a maximum flow of value $K$ in an uncapacitated network with $m$ edges can be found in $O(mn \sqrt{K} + K \log^2(m))$ time, implying an $O(nm)$ algorithm if $W = O(\min\{n, \frac{m}{\log(m)}\})$.

More details about algorithms of Feder, the reduction to the minimum cut problem and missing proofs from this section are given in the full version of the paper.

It is important to note that none of the theorems in this section require any additional assumptions about the weight function $w$.

## 5 Computing a Heavy Sequence

We first show a very simple $O(mMWPM)$ algorithm for computing a heavy sequence where $MWPM$ is the time complexity of finding a maximum weight perfect matching. Then we improve its time complexity to either $O(mn \log(n))$ or $O(mn + \sqrt{mn} \log(Wn))$ depending on whether we use classical $O(mn \log(n))$ algorithm [22] or $O(\sqrt{mn} \log(Wn))$ algorithm by Gabow and Tarjan [4] for finding a maximum weight perfect matching.

We first compute a maximal sequence of strongly stable matchings $S = (M_0, M_1, \ldots, M_z)$. Recall that from Lemma 15 given a strongly stable matching $M_i$ we can find a heavy matching $M'_i \in [M_i]$ with a single maximum weight perfect matching computation. We simply apply Lemma 15 to each of the matchings $M_0, M_1, \ldots, M_z$ and obtain a heavy maximal sequence of strongly stable matchings $M'_0, M'_1, \ldots, M'_z$. Such an algorithm obviously works in $O(mMWPM)$ time.

Let us now discuss Algorithm 1. We first compute a maximal sequence of strongly stable matchings $S = (M_0, M_1, \ldots, M_z)$. Then we find a heavy matching $M'_0 \in [M_0]$ using Lemma 15. In the next step we construct graphs $K_{\rho_i}$ where $\rho_i = \rho([M_i], [M_{i+1}])$ for each $0 \leq i < z$. Then for each $i$ we compute a maximum weight perfect matching of $K_{\rho_i}$. It can be easily proven that each edge of $G$ may appear only in one of the graphs $K_{\rho_i}$, thus the following holds: $|E(K_{\rho_0})| + |E(K_{\rho_1})| + \ldots + |E(K_{\rho_{z-1}})| = O(m)$ and overall it takes either $O(\min\{n, \frac{m}{\log(m)}\})$ or $O(mn + \sqrt{mn} \log(Wn))$ time to compute all maximum weight matchings.

With the aid of Lemma 18 we can construct a heavy maximal sequence of strongly stable matchings $(M'_0, M'_1, \ldots, M'_z)$. In order to do this we simply compute a heavy matching $M'_i$ based on previously computed $M'_{i-1}$ and a maximum weight matching of $K_{\rho_i}$. 

ISAAC 2018
References


Approximation Algorithm for Vertex Cover with Multiple Covering Constraints

Eunpyeong Hong
Department of Computer Science and Information Engineering,
National Taiwan University, Taipei, Taiwan
ephong93@gmail.com

Mong-Jen Kao
Department of Computer Science and Information Engineering,
National Chung-Cheng University, Chiayi, Taiwan
mjkao@cs.ccu.edu.tw

Abstract
We consider the vertex cover problem with multiple covering constraints in hypergraphs. In this problem, we are given a hypergraph $G = (V, E)$ with a maximum edge size $f$, a cost function $w : V \rightarrow \mathbb{Z}^+$, and edge subsets $P_1, P_2, \ldots, P_r$ of $E$ along with covering requirements $k_1, k_2, \ldots, k_r$ for each subset. The objective is to find a minimum cost subset $S$ of $V$ such that, for each edge subset $P_i$, at least $k_i$ edges of it are covered by $S$. This problem is a basic yet general form of classical vertex cover problem and a generalization of the edge-partitioned vertex cover problem considered by Bera et al.

We present a primal-dual algorithm yielding an $(f \cdot H_r + H_r)$-approximation for this problem, where $H_r$ is the $r$th harmonic number. This improves over the previous ratio of $(3c f \log r)$, where $c$ is a large constant used to ensure a low failure probability for Monte-Carlo randomized algorithms. Compared to previous result, our algorithm is deterministic and pure combinatorial, meaning that no Ellipsoid solver is required for this basic problem. Our result can be seen as a novel reinterpretation of a few classical tight results using the language of LP primal-duality.

2012 ACM Subject Classification Mathematics of computing → Approximation algorithms

Keywords and phrases Vertex cover, multiple cover constraints, Approximation algorithm

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.43

Funding This work is supported in part by Ministry of Science and Technology (MOST), Taiwan, under Grants MOST107-2218-E-194-015-MY3 and MOST106-2221-E-001-006-MY3.

1 Introduction

The vertex cover problem is one of the most well-known and fundamental problem in graph theory and approximation algorithms. Given an undirected hypergraph $G = (V, E)$ and a cost function $w : V \rightarrow \mathbb{Z}^+$, the objective is to find a minimum cost subset $S \subseteq V$ such that any edge in $E$ is incident to some vertex in $S$.

This problem is known to be NP-hard, and $f$-approximation algorithms based on simple LP rounding and LP primal-duality are known for this problem [10], where $f$ is the maximum size of the hyperedges. Assuming the unique game conjecture, approximating this problem to a ratio better than $(f - \epsilon)$ is NP-hard for any $\epsilon > 0$ [8].

The partial vertex cover problem is a natural generalization of the vertex cover problem. In this problem, we are given an additional parameter $k$ which is called the covering requirement. The objective of this problem is to find a minimum cost subset of $V$ which covers at least $k$ edges in $E$, i.e., at least $k$ edges of $E$ are incident to at least one vertex in $S$. 

© Eunpyeong Hong and Mong-Jen Kao; licensed under Creative Commons License CC-BY

Editors: Wen-Lian Hsu, Der-Tsai Lee, and Chung-Shou Liao; Article No. 43; pp. 43:1–43:11

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
Various methods have been developed to obtain tight approximations for this problem. Bshouty and Burroughs [3], who first proposed this problem, provided a 2-approximation algorithm for graphs, i.e., the case for which \( f = 2 \), using LP-rounding. Their algorithm generates \( |V| \) candidate covers each of which is constructed by guessing the most expensive vertex used in the optimal solution. Gandhi et al. [6] used the same technique to develop a primal-dual method which yields an \( f \)-approximation for hypergraphs. Mestre [9] used a more clever way to guess the most expensive vertex and improved the time complexity of the above algorithms.


Bera et al. [2] considered a generalization of the partial vertex cover problem for which they called the partition vertex cover problem. In this problem, we are given a partition \( E_1, E_2, \ldots, E_r \) of the edges along with covering requirements \( k_1, k_2, \ldots, k_r \). The objective is to find a minimum cost vertex subset that covers at least \( k_i \) edges of \( E_i \) for each \( 1 \leq i \leq r \).

They obtained a \((6c \log r)\)-approximation for normal graphs, where \( c \) is a large constant used for Monte-Carlo randomized algorithms to ensure low error probability. They used randomized iterative rounding on a strong LP which is derived by knapsack inequalities on the natural LP. This approach generalizes to hypergraphs with an approximation guarantee of \((3cf \log r)\). They also showed that, even for normal graph for which \( f = 2 \), it is NP-hard to approximate this problem to a ratio better than \( H_r \), which means \( O(f) \)-approximation for this problem is unlikely to exist.

Wolsey [11] proposed the submodular set cover problem, which is a general formulation to the above covering problems, and presented an \( H(\max_{S \subseteq S} g(S)) \)-approximation, where \( g \) is the input submodular function and \( S \) is the ground set. Chuzhoy et al. [4] presented a simpler analysis to obtain a similar result. Fujito [5] presented a primal-dual algorithm for this problem which is useful for some special cases such as the partial vertex cover problem.

**Our Focus and Contributions**

In this paper, we consider the vertex cover problem with multiple covering constraints (VC-MCC) in hypergraphs. In this problem, we are given a hypergraph \( G = (V, E) \), a cost function \( w : V \rightarrow \mathbb{Z}^+ \), and a number of covering constraints \( (P_1, k_1), (P_2, k_2), \ldots, (P_r, k_r) \), where each \( P_i \subseteq E \) is a subset of \( E \) and \( k_i \in \mathbb{Z}^+ \) is the covering requirement for \( P_i \). The objective is to find a minimum cost subset \( S \subseteq V \) such that, for each \( 1 \leq i \leq r \), at least \( k_i \) edges of \( P_i \) are covered by \( S \).

This problem is a basic yet general form of classical vertex cover and a further generalization of the edge-partitioned vertex cover problem considered in [2].

In this paper, we present a primal-dual algorithm that yields an \((f \cdot H_r + H_r)\)-approximation for this problem, improving over the previous ratio of \((cf \log r)\) due to [2]. Our main contribution is the following theorem.

**Theorem 1.** There is a deterministic \((f \cdot H_r + H_r)\)-approximation algorithm for VC-MCC which runs in polynomial time, where \( H_r \) is the \( r \)th harmonic number.

Compared to the previous result of \((cf \log r)\), our algorithm is deterministic and pure combinatorial, which means that our algorithm does not rely on heavy Ellipsoid LP solvers for this basic problem. Considering the lower-bound of \( H_r \) on the approximation ratio due to [2] and the well-known lower-bound of \( f \) for vertex cover, our result is much closer to the tight extent possible.
The novelty of this work lies in the way how we handle the dual variables. In contrast to previously known primal-dual approaches for covering problems, in which the dual variables can be handled freely, our approach manages the dual solutions carefully so that the following two criteria are met.

1. During the process, the cost of any vertex to be opened in the future must only be paid by the dual values possessed by the current unfulfilled covering constraints.
2. The overall dual value possessed by any unfulfilled covering constraint remains the same all the time.

This makes the approximation guarantee of $\log r$ possible.

Our result can be seen as a novel combination of the classical tight approximations with guarantees $f$ and $H_r$ for the covering problem, using the language of LP primal-duality.

Our ingredient includes the strong LP relaxation due to [2], which is derived by applying Knapsack-cover inequalities to the natural LP. We would like to remark, however, that the usage of strong LP relaxation in our result is not a necessity but rather a better and more intuitive exposition of our ideas on how the dual variables can be managed, and obtaining the same result using natural LP is possible.

Organization of this paper

The rest of this paper is organized as follows. In Section 2, we define the notations we will be using throughout this paper and introduce the strong LP formulations. We present our approximation algorithm in Section 3 and conclude with future directions in Section 4.

2 Preliminary

We use $G = (V,E)$ to denote a hypergraph $G$ with a vertex set $V$ and an edge set $E \subseteq 2^V$. Note that, under this notion, any edge $e \in E$ is a subset of $V$ that consists the incident vertices of the edge $e$. We use $f_G$ to denote the maximum cardinality of the edges in $G$, i.e., $f_G = \max_{e \in E} |e|$. The subscript $G$ is omitted when no ambiguity is there in the context.

For any edge subset $M \subseteq E$ and any vertex $v \in V$, we use $M(v)$ to denote the set of edges in $M$ that are incident to $v$, i.e., $M(v) := \{ e \in M : v \in e \}$. For any subset $A \subseteq V$, we use $M(A)$ to denote the set of edges in $M$ that are incident to the vertices in $A$, i.e., $M(A) = \bigcup_{v \in A} M(v)$.

Vertex Cover with Multiple Covering Constraints

In this problem, we are given a hypergraph $G = (V,E)$, a cost function $w : V \to \mathbb{Z}^+$, and a number of covering constraints $(P_1, k_1), (P_2, k_2), \ldots, (P_r, k_r)$, where for each $1 \leq i \leq r$, $P_i \subseteq E$ is a subset of $E$ and $k_i \in \mathbb{Z}^+$ is the covering requirement for $P_i$ to be fulfilled.\(^1\)

The objective of this problem is to find a vertex subset $S \subseteq V$ of minimum cost such that $|P_i(S)| \geq k_i$ for each $1 \leq i \leq r$.

Intuitively, this problem asks for a minimum cost subset such that in each $P_i$, at least $k_i$ edges are covered. A natural LP relaxation for this problem is given in Figure 1.

We have two sets of indicator variables in this LP formulation: For each $v \in V$, $x_v$ denotes the inclusion of $v$ into the cover and $y_e$ for each $e \in E$ indicates the coverage of $e$ by the

\(^1\) Without loss of generality, we assume that $k_i \leq |P_i|$ for all $1 \leq i \leq r$. 

ISAAC 2018
vertices chosen in the cover. The first inequality models the coverage of each edge $e \in E$ and the second inequality models the covering requirement for each $(P_i, k_i)$, $1 \leq i \leq r$.

However, the integrality gap of the natural LP can be arbitrarily large. This is illustrated by the following simple example. Consider a star with $d + 1$ vertices. Suppose that the cost of every vertex is 1 and we only have one constraint consisting of all edges with covering requirement 1. The optimal integral cost for this example is 1 while its optimal fractional cost is $1/d$, resulting a gap of $d$ which can be arbitrarily large.

**A Strong LP Relaxation**

Instead of using the natural LP relaxation, we use a strong LP relaxation due to [2], which is derived by applying Knapsack-cover inequalities to the natural LP given above. For any vertex subset $A \subseteq V$ and any $1 \leq i \leq r$, define

$$k_i(A) := \max \left\{ k_i - |P_i(A)|, \ 0 \right\}.$$  

Intuitively, $k_i(A)$ denotes the residue covering requirement to be fulfilled for $P_i$, if the vertex set $A$ were already chosen as part of the cover. For any vertex $v \in V \setminus A$, define

$$\beta_i(v, A) := \min \left\{ |P_i(v) \setminus P_i(A)|, \ k_i(A) \right\}.$$  

Intuitively, $\beta_i(v, A)$ is the amount of covering requirement $v$ can be fulfilled for $P_i$ if $A$ is already chosen as part of the cover. Clearly, $\beta_i(v, A)$ will be either $k_i(A)$ or the number of incident edges of $v$ in $P_i \setminus P_i(A)$, which is $|P_i(v) \setminus P_i(A)|$.

The strong LP relaxation we consider is as follows:

\[
\begin{align*}
\min & \sum_{v \in V} w_v x_v \\
\text{s.t.} & \sum_{v \in V \setminus A} \beta_i(v, A) \cdot x_v \geq k_i(A), \quad \forall 1 \leq i \leq r, \ \forall A \subseteq V \\
& x_v \geq 0, \quad \forall v \in V.
\end{align*}
\]

To see that LP-(S) gives a valid relaxation for VC-MCC, consider any feasible integral solution $\hat{x}$. It suffices to show that $\hat{x}$ is also contained in the feasible region of LP-(S).

- **Figure 1** A natural LP relaxation for VC-MCC.
Consider an arbitrary subset \( A \) of \( V \) and any constraint \( 1 \leq i \leq r \). Clearly, \( \hat{x} \) must remain feasible even if the vertices of \( A \) were already chosen as part of the cover in advance for free. Hence, the number of edges \( \hat{x} \) covers for \( P_i \) is at least \( k_i(A) \), and the inequality
\[
\sum_{v \in V \setminus A} \beta_i(v, A) \cdot x_v \geq k_i(A)
\]
must hold.

To see that LP-(S) is indeed a stronger relaxation than LP-(N), let us consider the simple star example and the inequality with respect to \( A = \emptyset \). Clearly, \( \beta_i(v, A) = 1 \) for all vertices \( v \) in this star. As a result, we have a constraint \( \sum_{v \in V} x_v \geq 1 \), and the optimal fractional solution will also be 1.

### The Dual LP for LP-(S)

In this paper we will be working around the dual LP of LP-(S), which is given as follows. In the following section we describe the algorithm in details. In §3.2 we establish the approximation guarantee.

#### 3 Our Approximation Algorithm for VC-MCC

In this section, we present our approximation algorithm for VC-MCC. Given an instance \( \Pi = (G = (V, E), w, (P_i, k_i)_{1 \leq i \leq r}) \) of VC-MCC, the algorithm will compute a series of feasible LP solutions of \( \Pi \) to LP-Dual-(S). During this process, a feasible cover for \( \Pi \) will gradually be formed. The approximation guarantee is then established by comparing the cost of the cover to the values of the dual solutions the algorithm computes.

In the following section we describe the algorithm in details. In §3.2 we establish the approximation guarantee.

#### 3.1 The Algorithm

The algorithm takes as input an instance \( \Pi \) of VC-MCC and outputs a feasible cover \( S \) for \( \Pi \).

Initially, \( S \) is set to be an empty set. In addition, the algorithm will maintain a set \( K \) which contains the set of covering constraints that have not been satisfied yet. The set \( K \) is initialized to be \( \{1, 2, \ldots, r\} \).

In the following, we first describe our primal-dual process. Then we describe how this primal-dual process can be transformed into a polynomial-time algorithm.

Our primal-dual process, denoted PD-VC-MCC, starts with a trivial dual solution for which \( z_{i,A} = 0 \) for all \( 1 \leq i \leq r \) and all \( A \subseteq V \). In each iteration, it proceeds as follows:

1. It raises \( z_{i,S} \) at the rate of \( 1/k_i(S) \) for all \( i \in K \) until the Inequality (*) of some vertex in LP-Dual-(S), say, \( v \in V \setminus S \), becomes tight. Then it adds the vertex \( v \) to the set \( S \).
Input An instance II of VC-MCC  
Output A feasible cover $S$

1. $S \leftarrow \emptyset$, $K \leftarrow \{1, \ldots, r\}$ and $z_{i,A} \leftarrow 0$ for all $1 \leq i \leq r$ and all $A \subseteq V$.
2. Repeat until $K$ becomes an empty set.
   a. Raise $z_{i,S}$ for all $i \in K$ simultaneously at the rate of $1/k_i(S)$ until the inequality (*) in LP-Dual-(S) for some $v \in V \setminus S$ becomes tight.
   b. Add the vertex $v$ into $S$.
   c. For every $i \in K$ such that $k_i(S) = 0$, $K \leftarrow K \setminus \{i\}$ and $z_{i,A} \leftarrow 0$ for all $A \subseteq V$.
3. Return $S$

**Figure 2** A formal description of our primal-dual process PD-VC-MCC.

2. For each constraint $i \in K$ with $k_i(S)$ becoming zero after $v$ is added to $S$, our primal-dual process:
   a. sets $z_{i,A}$ to be zero for all $A \subseteq V$ and
   b. removes $i$ from $K$.
This process repeats until the set $K$ becomes empty. Then $S$ is returned as the approximate solution. A high-level pseudo-code of this primal-dual process is given in Figure 2 for further reference.

We remark that, the step 2.(a) above of resetting $z_{i,A}$ to zero for all $A \subseteq V$ when $i$ is to be removed from $K$ is very important and is the key to obtain a guarantee of $H_r$. The reason is that it allows the overall contribution of the remaining covering constraints to remain balanced.

Our approximation algorithm, denoted Approx-VC-MCC, mimics the operations of the above primal-dual process. Instead of maintaining the dual variables, it keeps track of the slack of the Inequality (*) in LP-Dual-(S) for each vertex, i.e., the amount before it becomes tight.

For each $v \in V$, let $\tilde{w}_v$ denote the slack of the vertex constraint $v$ before it becomes tight. Initially, $\tilde{w}_v$ is set to be $w_v$. We need a notion that reflects the raising process of the dual variables. For each $v \in V$ and $A \subseteq V \setminus \{v\}$, define

$$s(v, A) := \sum_{i \in K} \frac{\beta_i(v, A)}{k_i(A)}.$$  

Intuitively, $s(v, A)$ denotes the speed for which $\tilde{w}_v$ will decrease if we raised the dual variables $z_{i,A}$ at the speed of $1/k_i(A)$ for all $i \in K$.

Furthermore, in order for the update of $\tilde{w}_v$ for each $v \in V$ to proceed, we use $\Phi_{v,i}$ to denote the contribution of dual variables $z_{i,A}$ for all possible $A$ towards $\tilde{w}_v$. $\Phi_{v,i}$ is initialized to be zero for all $v \in V$ and $1 \leq i \leq r$.

Now we formally describe our approximation algorithm. In each iteration, the algorithm finds the among the vertices in $V \setminus S$ the one with the smallest ratio of $\tilde{w}_v/s(v, A)$. Formally speaking, it computes

$$v = \arg\min_{u \in V \setminus S} \frac{\tilde{w}_u}{s(u, A)} \quad \text{and} \quad t_v = \frac{\tilde{w}_v}{s(v, A)}.$$
Intuitively, \( v \) is the first vertex constraint to become tight in this iteration in the primal-dual process and \( t_v \) is the corresponding amount of time it takes.

Then the algorithm proceeds as follows:

1. For each \( u \in V \setminus S \), the algorithm:
   a. updates \( \hat{w}_u \) by setting \( \hat{w}_u \leftarrow \hat{w}_u - s(u, S) \cdot t_v \).
   b. update the contribution \( \Phi_{u,i} \) for each \( i \in K \) by setting \( \Phi_{u,i} \leftarrow \Phi_{u,i} + \beta_{i}(u, S) \cdot t_v \).
2. Add \( v \) to the set \( S \).
3. For each \( i \in K \) such that \( k_i(S) \) is zero, the algorithm does the following:
   a. Update \( \hat{w}_u \) for all \( u \in V \setminus S \) by setting \( \hat{w}_u \leftarrow \hat{w}_u + \Phi_{u,i} \).
   b. Remove \( i \) from \( K \).

The algorithm repeats until the set \( K \) becomes empty. Then \( S \) is returned as the approximate solution.

### 3.2 Analysis

In this section, we provide the analysis of our approximation algorithm Approx-VC-MCC and prove Theorem 1. First we show that our algorithm always terminates and returns a feasible cover. Then we establish the approximation guarantee.

#### Feasibility of algorithm Approx-VC-MCC

We first establish the feasibility of our primal-dual process. Then we argue that algorithm Approx-VC-MCC does mimic the execution of this process and runs in polynomial time.

\begin{itemize}
  \item \textbf{Lemma 2.} The primal-dual process PD-VC-MCC always terminates and returns a feasible cover.
\end{itemize}

\textbf{Proof.} Since PD-VC-MCC only terminates when the set \( K \) becomes empty and it finds a feasible cover, it suffices to argue that PD-VC-MCC always terminates, provided that there is a feasible cover for the input instance.

Assume for contradiction that the input instance has a feasible solution but PD-VC-MCC does not terminate. Consider the set \( S \) the process currently has. The process runs eternally since no vertex \( v \in V \setminus S \) becomes tight as \( z_{i,S} \) is constantly raising for all \( i \in K \). This implies that \( \beta_{i}(v, S) = |P_i(v) \setminus P_i(S)| = 0 \) for all \( v \in V \setminus S \) and all \( i \in K \).

This means that all the edges have already been covered by \( S \), a contradiction. \hfill \Box

To see that algorithm Approx-VC-MCC simulates the execution of PD-VC-MCC, it suffices to observe that

\begin{itemize}
  \item \( \hat{w}_v \) records the slack \( w_v = \sum_{1 \leq i \leq r, A \subseteq V \setminus \{v\}} \beta_i(v, A) \cdot z_{i,A} \) of the constraint (*) for all \( v \in V \setminus S \) during all iterations,
  \item \( \Phi_{v,i} \) keeps track of the value \( \sum_{A \subseteq V \setminus \{v\}} \beta_i(v, A) \cdot z_{i,A} \) so that it can be used to reflect the operation of resetting \( z_{i,A} \) to zero for \( i \) that is about to be removed from \( K \).
\end{itemize}

We have the following lemma.

\begin{itemize}
  \item \textbf{Lemma 3.} Algorithm Approx-VC-MCC mimics the execution of primal-dual process PD-VC-MCC and runs in polynomial time.
\end{itemize}

Lemma 2 and Lemma 3 establish the feasibility of algorithm Approx-VC-MCC.
Approximation Guarantee

To establish the approximation guarantee, we compare the cost of the solution our algorithm returns to the values of the dual solutions our primal-dual process maintains, which will be valid lower-bounds for the cost of optimal solutions by the weak LP duality.

Let \( S = \{v_1, v_2, \ldots, v_m\} \) denote the cover returned by the algorithm, where the indices of the vertices denote the order for which they are added to the set \( S \). For any \( 0 \leq j \leq m \), we use \( A_j \) to denote the set of the first \( j \) vertices that are added to \( S \), i.e., \( A_j := \{v_1, v_2, \ldots, v_j\} \).

Without loss of generality, we also assume that the covering constraints \( P_1, P_2, \ldots, P_r \) are fulfilled by the algorithm in this order.

For any \( 1 \leq i \leq r \), let \( \pi(i) \) denote the index for which the inclusion of \( v_{\pi(i)} \) into \( S \) fulfills \( P_i \). Consider the moment when \( P_i \) is just fulfilled, i.e., when \( i \) was removed from \( K \) by the algorithm, and \( z_{i,A} \) has not yet been reset. Let \( \tilde{z}^{(i)} \) denote the dual solution the algorithm maintains at this moment, and \( \text{Val}(\tilde{z}^{(i)}) \) denote the objective value of \( \tilde{z}^{(i)} \). It follows that

\[
\text{Val}(\tilde{z}^{(i)}) := \sum_{1 \leq t \leq r, \ A \subseteq V} k_t(A) \cdot \tilde{z}_{t,A}^{(i)} = \sum_{i \leq t \leq r} \sum_{0 \leq j < \pi(i)} k_t(A_j) \cdot \tilde{z}_{t,A_j}^{(i)},
\]

where the second equality holds since our algorithm resets \( \tilde{z}_{t,A}^{(i)} \) to zero for all \( 1 \leq t < i \) and all \( A \subseteq V \).

In the above equality we write \( \text{Val}(\tilde{z}^{(i)}) \) as the sum of dual values each unfulfilled covering requirement possesses. The following lemma says that the dual value possessed by each unfulfilled constraint is the same.

\[\textbf{Lemma 4.}\] For any \( 1 \leq i \leq r \) and any \( t_1, t_2 \) with \( i \leq t_1 \neq t_2 \leq r \), we have

\[
\sum_{0 \leq j < \pi(i)} k_{t_1}(A_j) \cdot \tilde{z}_{t_1,A_j}^{(i)} = \sum_{0 \leq j < \pi(i)} k_{t_2}(A_j) \cdot \tilde{z}_{t_2,A_j}^{(i)}.
\]

\[\textbf{Proof.}\] This lemma follows directly from the way our primal-dual approach handles the dual variables. Since \( k_t(S) \) only changes when a new vertex becomes tight and since we always raise \( z_{i,S} \) for each \( i \in K \) at the rate of \( 1/k_t(S) \), the total dual value possessed by any \( P_i \) with \( i \in K \) will be the same. \( \square \)

In the following we analyze the cost of \( S \) and relate it to the dual values of \( \tilde{z}^{(i)} \) the algorithm maintains for all \( 1 \leq i \leq r \).

Consider a vertex \( v \in S \) and the moment when \( v \) just becomes tight. Suppose that at that time, the algorithm has already fulfilled \( t \) covering constraints. Then, from the Inequality (*) of LP-Dual-(S), it follows that

\[
w_v = \sum_{1 \leq i \leq r, \ A \subseteq V \setminus \{v\}} \beta_i(v, A) \cdot z_{i,A} = \sum_{t < i \leq r, \ A \subseteq V \setminus \{v\}} \beta_i(v, A) \cdot z_{i,A},
\]

where the second equality holds since, by design, our algorithm has already reset \( z_{i,A} \) to zero for all \( 1 \leq i \leq t \) and all \( A \subseteq V \) before \( v \) becomes tight.

For each \( t < i \leq r \), define \( \Phi_{v,i} := \sum_{A \subseteq V \setminus \{v\}} \beta_i(v, A) \cdot z_{i,A} \). Then we have

\[
w_v = \sum_{t < i \leq r} \Phi_{v,i}.
\]

Intuitively, \( \Phi_{v,i} \) is the share for which the covering constraint \( i \) contributes towards the cost of vertex \( v \). We will charge the cost of \( v \) to the covering constraints \( P_t \) for all \( t < i \leq r \), each of which gets a charge of \( \Phi_{v,i} \).
For each $1 \leq i \leq r$, let $\text{cost}(P_i)$ denote the total charge $P_i$ receives from the vertices in $S$. We have the following lemma, which bounds $\text{cost}(P_i)$ using the dual value it possesses in $\hat{z}^{(i)}$.

**Lemma 5.** For any $1 \leq i \leq r$, we have

$$\text{cost}(P_i) \leq (f + 1) \cdot \sum_{0 \leq j < \pi(i)} k_i(A_j) \cdot \hat{z}^{(i)}_{i,A_j}.$$  

**Proof.** From the definition of $\text{cost}(P_i)$, only $v_1, v_2, \ldots, v_{\pi(i)}$ will charge $P_i$, and it follows that

$$\text{cost}(P_i) = \sum_{1 \leq t \leq \pi(i)} \Phi_{v_t,i} = \sum_{1 \leq t \leq \pi(i)} \sum_{A \subseteq \bigcap\{v_t\}} \beta_t(v_t, A) \cdot \hat{z}^{(i)}_{i,A} = \sum_{1 \leq t < \pi(i)} \sum_{0 \leq j < t} \beta_t(v_t, A_j) \cdot \hat{z}^{(i)}_{i,A_j} = \sum_{0 \leq j < \pi(i)} \sum_{j < t < \pi(i)} \beta_t(v_t, A_j) \cdot \hat{z}^{(i)}_{i,A_j}.$$  

Hence, to prove this lemma, it suffices to show that

$$\sum_{0 \leq j < \pi(i)} \sum_{j < t < \pi(i)} \beta_t(v_t, A_j) \cdot \hat{z}^{(i)}_{i,A_j} \leq (f + 1) \cdot \sum_{0 \leq j < \pi(i)} k_i(A_j) \cdot \hat{z}^{(i)}_{i,A_j}. \quad (2)$$

To prove Ineq. (2), we first prove the following inequality:

$$\sum_{0 \leq j < \pi(i)} \sum_{j < t < \pi(i)} \beta_t(v_t, A_j) \cdot \hat{z}^{(i)}_{i,A_j} \leq f \cdot \sum_{0 \leq j < \pi(i)} k_i(A_j) \cdot \hat{z}^{(i)}_{i,A_j} \quad (3)$$

Compare the l.h.s. and the r.h.s. of (3), it suffices to argue that

$$\sum_{j < t < \pi(i)} \beta_t(v_t, A_j) \leq f \cdot k_i(A_j) \quad \text{for all } 0 \leq j < \pi(i).$$

Consider any fixed $j$ with $0 \leq j < \pi(i)$ and any $t$ with $j < t < \pi(i)$. Since $v_t$ is not the vertex whose inclusion into $S$ fulfills $P_i$, it follows that $|P_i(v_t) \setminus P_i(A_j)| < k_i(A_j)$, and hence $\beta_t(v_t, A_j) = |P_i(v_t) \setminus P_i(A_j)|$.

Furthermore, under the condition that $A_j$ has already been chosen, the inclusion of $\{v_{j+1}, v_{j+2}, \ldots, v_{\pi(i)-1}\}$ into $S$ does not fulfill $P_i$.

This implies that $\left| \bigcup_{j \leq t < \pi(i)} (P_i(v_t) \setminus P_i(A_j)) \right| < k_i(A_j)$. Therefore, we have

$$\sum_{j < t < \pi(i)} \beta_t(v_t, A_j) = \sum_{j < t < \pi(i)} \left| P_i(v_t) \setminus P_i(A_j) \right| \leq f \cdot \left| \bigcup_{j < t < \pi(i)} (P_i(v_t) \setminus P_i(A_j)) \right| < f \cdot k_i(A_j),$$

where the second last inequality holds since the size of each hyperedge is at most $f$.

This proves Ineq. (3).
Second, observe that the following inequality holds:
\[
\sum_{0 \leq j < \pi(i)} \beta_i(v_{\pi(i)}, A_j) \cdot \hat{z}^{(i)}_{i,A_j} \leq \sum_{0 \leq j < \pi(i)} k_i(A_j) \cdot \hat{z}^{(i)}_{i,A_j},
\]
(4)

since \(\beta_i(v_{\pi(i)}, A_j) \leq k_i(A_j)\) by the definition of \(\beta_i(v_{\pi(i)}, A_j)\).

From Ineq. (3) and Ineq. (4), the Inequality (2) is proved and this lemma holds. \(\blacksquare\)

In the following we establish the approximation guarantee of our algorithm.

\textbf{Lemma 6.}

\[\text{cost}(S) \leq (f + 1) \cdot H_r \cdot \text{OPT},\]

where \(H_r\) is the \(r\)th harmonic number and \(\text{OPT}\) is the cost of any optimal solution.

\textbf{Proof.} By Lemma 5 and the definition of \(\text{cost}(P_i)\), we have
\[
\text{cost}(S) = \sum_{1 \leq i \leq r} \text{cost}(P_i) \leq (f + 1) \cdot \sum_{1 \leq i \leq r} \sum_{0 \leq j < \pi(i)} k_i(A_j) \cdot \hat{z}^{(i)}_{i,A_j}.
\]

By Lemma 4, for each \(1 \leq i \leq r\), we have
\[
\sum_{0 \leq j < \pi(i)} k_i(A_j) \cdot \hat{z}^{(i)}_{i,A_j} = \frac{1}{r - i + 1} \cdot \text{Val}(\hat{z}^{(i)})
\]

Since each \(\hat{z}^{(i)}\) is a feasible dual solution for LP-Dual-(S), we have \(\text{Val}(\hat{z}^{(i)}) \leq \text{OPT}\) for all \(1 \leq i \leq r\), and
\[
\text{cost}(S) \leq (f + 1) \cdot \sum_{1 \leq i \leq r} \frac{1}{r - i + 1} \cdot \text{OPT} \leq (f + 1) \cdot H_r \cdot \text{OPT}
\]
as claimed. \(\blacksquare\)

\section{Conclusion}

We conclude with future directions and open problems. First, considering the lower-bounds of \(H_r\) and \(f\) for this problem, our \((f \cdot H_r + H_r)\)-approximation ratio has an extra \(H_r\) factor in it. However, it seems unclear how this excess \(H_r\) factor can be dropped.

Although the approaches of [5, 6, 9] can be used to obtain tight \(f\)-approximation for the partial vertex cover problem, it seems difficult to adopt their techniques to our problem. The reason is that, when multiple covering constraints exist, it seems intricate how the key properties of their approaches can be ensured simultaneously for each covering constraint. We believe that this would be an interesting direction to explore.

Second, clarifying the exact lower bound of approximation ratio for this problem is also interesting. For now, \(\max(H_r, f)\) is what we only know.

\textbf{References}


Correlation Clustering Generalized

David F. Gleich\(^1\)
Department of Computer Science, Purdue University, West Lafayette, Indiana, USA
dgleich@purdue.edu

Nate Veldt\(^2\)
Department of Mathematics, Purdue University, West Lafayette, Indiana, USA
lvdlt@purdue.edu

Anthony Wirth\(^3\)
School of Computing and Information Systems, The University of Melbourne, Parkville, Victoria, Australia
awirth@unimelb.edu.au

Abstract

We present new results for LambdaCC and MotifCC, two recently introduced variants of the well-studied correlation clustering problem. Both variants are motivated by applications to network analysis and community detection, and have non-trivial approximation algorithms.

We first show that the standard linear programming relaxation of LambdaCC has a $\Theta(\log n)$ integrality gap for a certain choice of the parameter $\lambda$. This sheds light on previous challenges encountered in obtaining parameter-independent approximation results for LambdaCC. We generalize a previous constant-factor algorithm to provide the best results, from the LP-rounding approach, for an extended range of $\lambda$.

MotifCC generalizes correlation clustering to the hypergraph setting. In the case of hyperedges of degree 3 with weights satisfying probability constraints, we improve the best approximation factor from 9 to 8. We show that in general our algorithm gives a $4(k-1)$ approximation when hyperedges have maximum degree $k$ and probability weights. We additionally present approximation results for LambdaCC and MotifCC where we restrict to forming only two clusters.

2012 ACM Subject Classification Mathematics of computing → Approximation algorithms

Keywords and phrases Correlation Clustering, Approximation Algorithms

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.44


1 Introduction

Correlation Clustering (CC), introduced by Bansal et al. [3], is often viewed as a partitioning problem on signed graphs. Given $n$ nodes whose edges have so-called positive or negative weights (maybe both), the goal is to find the clustering which correlates as much as possible with the edge weights. That is, a positive-weight edge suggests two nodes should be clustered together, while a negative-weight edge suggests separation, and these weights are

\[^1\] D.F.G. is supported by the DARPA Simplex program, the Sloan Foundation, and NSF awards IIS-154648, CCF-1149756 and CCF-093937.

\[^2\] N.V. supported by NSF award CCF-1149756.

\[^3\] A.W. is supported by the Melbourne School of Engineering.
Correlation Clustering Generalized

in some sense soft constraints. There is a variety of settings for Correlation Clustering, including different objective functions, and special classes of edge weights, leading to a rich and interesting family of approximation algorithms and hardness results.

In this document, we consider two recent variants of the problem, called Lambda Correlation Clustering (LambdaCC) [22] and Motif Correlation Clustering (MotifCC) [17]. Although introduced independently, both problems are motivated by applications to community detection in unsigned graphs, and are interesting to study from a theoretical perspective, each coming with non-trivial approximation guarantees. LambdaCC is a generalization of the standard unweighted CC in which all positive edges have a common weight, while all negative edges have another (possibly different) common weight. A parameter $\lambda$ determines these two weights and, implicitly, controls the size and structure of clusters formed by optimizing the objective. MotifCC is a generalization of Correlation Clustering to hypergraphs, designed to provide a framework for clustering graphs based on higher-order subgraph patterns (i.e., motifs). We present new results for LambdaCC and MotifCC, not only where the number of clusters formed is an outcome of minimizing the objective, but also where we (additionally) restrict to forming only two clusters. In summary, we make the following contributions:

1. We show that there exists some small $\lambda$ such that the LambdaCC LP relaxation has a $\Theta(\log n)$ integrality gap. This hints at why constant-factor approximations have been developed for $\lambda \geq 1/2$, but no analogous result has been found for small $\lambda$. We also extend the analysis of our previous algorithm for LambdaCC [22] to outline the range of $\lambda < 1/2$ values, that admit an approximation factor in $o(\log n)$.

2. We show that when we restrict to two clusters, LambdaCC reduces to the Min Uncut problem, which implies an $O(\sqrt{\log n})$ approximation for this special case [1].

3. We generalize the 4-approximation of Charikar et al. for complete unweighted correlation clustering to obtain a $4(k - 1)$ approximation for MotifCC on hypergraphs with edges of degree $k$ where edge weights satisfy probability constraints. We consider the same LP relaxation as Li et al. [17], and apply a similar rounding technique. However, we provide an approximation guarantee for arbitrary $k$ that is linear in $k$, in addition improving the factor for $k = 3$ from 9 to 8.

4. For Two-Cluster MotifCC, we design an algorithm that gives an asymptotic $1 + k 2^{k-2}$ approximation by generalizing the 3-approximation of Bansal et al [3] for 2-CC (which applies when $k = 2$). This is the first combinatorial result for 2-MotifCC, and is a 7-approximation for $k = 3$.

2 Background and Previous Results

In the most general formulation of Correlation Clustering on (undirected) graphs – excluding, for the moment, the generalization to hypergraphs – each pair of nodes $(i, j)$ is assigned a pair of nonnegative weights $(w_{ij}^+, w_{ij}^-)$, i.e., a similarity score and a dissimilarity score. In many cases, only one of these weights is assumed to be nonzero, to indicate strict similarity or strict dissimilarity between pairs of nodes. We focus on the objective of minimizing disagreements, which can be formally expressed as an integer linear program:

\[
\begin{align*}
\text{minimize} & \quad \sum_{i<j} w_{ij}^+ x_{ij} + w_{ij}^- (1 - x_{ij}) \\
\text{subject to} & \quad x_{ij} \leq x_{ik} + x_{jk} & \text{for all } i, j, k \\
& \quad x_{ij} \in \{0, 1\} & \text{for all } i < j
\end{align*}
\]

The variable $x_{ij}$ is 1 if nodes $i$ and $j$ are in separate clusters, and is 0 otherwise. Thus, a clustering that separates $i, j$ incurs a penalty (also called a mistake, or a disagreement) of
weight \( w_{ij}^+ \), while if \( i, j \) are together the penalty has weight \( w_{ij}^- \). The objective of maximizing agreements has also been extensively considered: it shares the same set of optimal clusterings as minimizing disagreements, but is easier from the perspective of approximations. For the general weighted case, correlation clustering is equivalent to Minimum Multicut \([10]\), which implies an \( O(\log n) \) approximation, but also suggests that Correlation Clustering (with general weights) is unlikely to be approximated to within a constant factor in polytime \([6]\). For weights satisfying probability constraints (i.e., \( w_{ij}^+ + w_{ij}^- = 1 \)), Ailon et al. gave a 2.5 approximation \([2]\). The best approximation factor for the standard unweighted problem (i.e., \( (w_{ij}^+, w_{ij}^-) \in \{(0,1), (1,0)\} \)) is slightly better than 2.06 \([7]\).

Fixing the number of clusters

In general, Correlation Clustering does not require a user to specify number of clusters to be formed; the number of clusters arises naturally by optimizing the objective. However, restricting the output of Correlation Clustering to a fixed number of clusters has also been studied extensively. In their seminal work, Bansal et al. showed a 3-approximation for minimizing disagreements in the two-cluster unweighted case (2-Correlation Clustering) \([3]\). Later, Giotis and Guruswami showed a polynomial time approximation scheme for maximizing agreements and for minimizing disagreements, when the number clusters is a fixed constant \([12]\). For the maximization version, 2-Correlation Clustering is equivalent to Max Cut; based on this Dasgupta et al. showed a 0.878-approximation for arbitrary weights \([9]\). Extending Bansal et al.’s approach, Coleman et al. introduced faster, greedy 2-approximations for minimizing disagreements for unweighted 2-Correlation Clustering \([8]\), and gave a more extensive overview of the historical interest in this problem. Given this recurring interest in correlation clustering with a fixed number of clusters, we address several questions involving the two-cluster case in this manuscript.

2.1 Lambda Correlation Clustering

In previous work, we introduced the LambdaCC objective, which can be viewed as a special case of weighted correlation clustering (1) in which \( (w_{ij}^+, w_{ij}^-) \in \{(1 - \lambda, 0), (0, \lambda)\} \) for some user-chosen parameter \( \lambda \in (0, 1) \). This provides the following framework for partitioning unsigned networks: given an unsigned graph \( G = (V, E) \), treat each edge, in \( E \), as a positive edge of weight \( 1 - \lambda \) in a signed graph, and treat each non-edge as a negative edge with weight \( \lambda \). When \( \lambda = 1/2 \), LambdaCC amounts to unweighted Correlation Clustering; with small \( \lambda \), LambdaCC amounts to Sparsest Cut; and when \( \lambda \) is large, LambdaCC amounts to Cluster Deletion. We previously outlined another, similar, edge-weighting scheme \([22]\) that is equivalent to the Modularity objective \([18]\). We do not consider it here, however, as this scheme does not appear to lead to new approximation results.

For \( \lambda > 1/2 \), we gave a 3-approximation based on the LP-rounding technique of van Zuylen and Williamson \([21]\), and a 2-approximation which holds specifically for \( \lambda > |E|/(1 + |E|) \), hence, for Cluster Deletion. We also note that when \( \lambda > 1/2 \), LambdaCC can be viewed as a specific case of the specially weighted correlation clustering variant considered by Puleo and Milenkovic \([19]\), for which they gave a 5-approximation based on a generalization of the LP rounding scheme of Charikar et al. \([5]\). However, the proof strategies for all of these algorithms fail when considering arbitrarily small \( \lambda \).
2.2 Motif Correlation Clustering

Li et al. introduced a higher-order generalization of Correlation Clustering, which they call Motif Correlation Clustering (MotifCC), as a means for clustering networks based on higher-order motif patterns shared among nodes [17]. This objective is motivated by previous successful results for motif-based graph clustering (see e.g., [4]). Although a similar higher-order correlation clustering objective was considered by Kim et al. for image segmentation [16], Li et al. were the first to study the objective from a theoretical perspective.

In their approach, we let $E_k$ denote the set of all $k$-tuples of nodes in $G$, and let each $E \in E_k$ have a positive weight, $w^+_E$, and a negative weight, $w^-_E$. If a clustering separates at least one pair of nodes in $E$, this gives a penalty of $w^+_E$; otherwise, there is a penalty of $w^-_E$. MotifCC is formally expressed as the following ILP, a generalization of ILP (1):

\[
\begin{align*}
\text{minimize} & \quad \sum_{E \in E_k} w^+_E x_E + w^-_E (1 - x_E) \\
\text{subject to} & \quad x_{uv} \leq x_{uw} + x_{vw} \quad \text{for all } u, v, w \\
& \quad x_{uv} \in \{0, 1\} \quad \text{for all } u < v \\
& \quad x_{uv} \leq x_E \quad \text{for all } u, v \in E \\
& \quad (k - 1)x_E \leq \sum_{u,v \in E} x_{uv} \quad \text{for all } E \in E_k \\
& \quad x_E \in \{0, 1\} \quad \text{for all } E \in E_k.
\end{align*}
\]

The first two constraints above ensure the variables encode a clustering ($x_{uv} = 1$ if $u, v$ are separated). Since $x_E$ is binary, constraint $x_E \geq x_{uv}$ ensures that if any two nodes $u, v$ in $E$ are separated, then $x_E = 1$ (i.e., the $k$-tuple is split). The fourth constraint guarantees that $x_E = 0$ if all pairs of nodes in $E$ are together. Li et al. considered an even more general objective, which they referred to as Mixed Motif Correlation Clustering (MMCC), where motifs of multiple sizes are considered at once, and the objective is a positive linear combination of objectives of the form (2) for different values of $k$. In their analysis they restrict to hyperedges of size 2 and 3, in other words they optimize an objective like this:

\[
\begin{align*}
\text{minimize} & \quad \sum_{u<v} w^+_uv x_{uv} + w^-uv (1 - x_{uv}) + \sum_{E \in E_3} w^+_E x_E + w^-_E (1 - x_E).
\end{align*}
\]

For this setting, they show a 9-approximation for the problem when hyperedge weights satisfy probability constraints ($w^+_E + w^-_E = 1$, for every hyperedge $E$ of size 2 or 3). Recently, Fukunga gave an $O(k \log n)$ approximation for general weighted hypergraphs by rounding the same LP [11].

3 New Results for LambdaCC

Given a signed graph, $G$, in which every pair of nodes is part of a negative edge set, $E^-$, or a positive edge set, $E^+$, the linear program relaxation of LambdaCC is

\[
\begin{align*}
\text{minimize} & \quad \sum_{(i,j) \in E^+} (1 - \lambda) x_{ij} + \sum_{(i,j) \in E^-} \lambda (1 - x_{ij}) \\
\text{subject to} & \quad x_{ij} \leq x_{ik} + x_{jk} \quad \text{for all } i, j, k \\
& \quad 0 \leq x_{ij} \leq 1 \quad \text{for all } i < j
\end{align*}
\]

Although a constant-factor approximation for LambdaCC exists for $\lambda \geq 1/2$, by rounding LP (3), we show that there exists some small $\lambda$ such that the integrality gap is $O(\log n)$. We then give parameter-dependent approximation guarantees for small $\lambda$, and consider new results for two-cluster LambdaCC.
3.1 Integrality Gap for the LambdaCC Linear Program

Demaine et al. prove that the integrality gap for the general weighted Correlation Clustering LP relaxation is $O(\log n)$ [10]. This does not immediately imply anything for our specially weighted case, but adapting some of their ideas, and adding some non-trivial steps, does reveal an $O(\log n)$ integrality gap for the LambdaCC linear program relaxation.

The proof takes the following steps.
1. Construct an instance of LambdaCC from an expander graph, $G$.
2. Prove that, because of the expander properties of $G$, the optimal LambdaCC clustering must make $\Omega(n)$ mistakes.
3. Demonstrate the LP relaxation has a feasible solution with a score of $O(n/\log n)$.

In order to accomplish third step listed above, we do not (necessarily) produce a feasible solution for the standard LP relaxation of LambdaCC: in particular, in our solution triangle constraints are not guaranteed. Instead, we produce a feasible solution for a related linear program considered by Wirth in his PhD thesis [23]. The fundamental construct of this LP is the Negative Edge with Positive Path Cycle (NEPPC), where, NEPPC($i_1, i_2, \ldots, i_m$) represents a sequence (a path) of (positive) edges, $(i_1, i_2), (i_2, i_3), \ldots, (i_{m-1}, i_m) \in E$, with a single (negative) non-edge completing the cycle: $(i_1, i_m) \notin E$. For LambdaCC, defined on a graph $G = (V, E)$, with parameter $\lambda \in (0, 1)$, we have the linear program:

$$\begin{align*}
\text{minimize} & \quad \sum_{(i,j) \in E} (1 - \lambda) x_{ij} + \sum_{(i,j) \notin E} \lambda (1 - x_{ij}) \\
\text{subject to} & \quad x_{ij}, i,j \in m \\
\text{for all} & \quad \text{NEPPC($i_1, i_2, \ldots, i_m$)} \\
\text{for all} & \quad (i, j) \notin E \\
\end{align*}$$

(4)

Wirth [23] proved that the set of optimal solutions to the NEPPC linear program (4) is exactly the same as the optimal solution set to the Correlation Clustering LP, the relation of ILP (1). Since a feasible solution for the LambdaCC NEPPC linear program (4) is an upper bound on the optimum for (4), which is the same as the optimum for the standard LambdaCC LP, we can bound the optimum of the latter. We now prove our result:

Theorem 1. There exists some $\lambda$ such that the integrality gap of LP (3) is $O(\log n)$.

Proof. The expander graph

Let $G = (V, E)$ be a $(d, c)$-expander graph, where both $d$ and $c$ are constants (Reingold et al. proved that such expanders exist [20]). That is, $G$ is $d$-regular, and for every $S \subset V$ with $|S| \leq n/2$, we have

$$\frac{\text{cut}(S)}{|S|} \geq c \quad \Rightarrow \quad \frac{\text{cut}(S)}{|S|} + \frac{\text{cut}(\bar{S})}{|\bar{S}|} \geq c \quad \Rightarrow \quad \frac{\text{cut}(S)}{|S||\bar{S}|} \geq \frac{c}{n}$$

where $\text{cut}(S)$ denotes the number of edges between $S$ and $\bar{S} = V \setminus S$. Define the scaled sparsest cut of a set $S$ to be $\text{cut}(S)/(|S||\bar{S}|)$ and let $\lambda^*$ minimize this ratio over all possible sets $S \subset V$. In previous work we showed that for any $\lambda \leq \lambda^*$, the optimal LambdaCC clustering places all nodes into one cluster, but there exists a range of $\lambda$ values slightly larger than $\lambda^*$ such that the optimum clustering coincides with a partitioning that produces the

\footnote{Although the proof is shown for the unweighted case, we note that all aspects of the proof immediately carry over to the weighted case.}
scaled sparsest cut score [22]. For the expander graph we consider, this $\lambda^*$ is at most the scaled sparsest cut score obtained by setting $S$ to be a single node, so we have these upper and lower bounds on $\lambda^*$: $c/n \leq \lambda^* \leq d/(n - 1)$.

The LambdaCC construction

Let $S^*$ be a set inducing an optimal scaled sparsest cut partition: $\lambda^* = \text{cut}(S^*)/(|S^*||\bar{S}^*|)$. From Theorem 3.2 in our previous work [22], we know that there exists some $\lambda'$, slightly larger than $\lambda^*$ whose optimum LambdaCC solution is the bipartition $\{S^*, \bar{S}^*\}$; let the LambdaCC score of this solution be $OPT$, and let $\epsilon = \lambda' - \lambda^*$. We can choose $\epsilon > 0$ to be arbitrarily small, so it suffices to assume $\lambda' < 2\lambda^*$.

Bounding OPT from below

With our choice of $\lambda'$, by definition,

$$OPT = \text{cut}(S^*) - \lambda' |S^*||\bar{S}^*| + \lambda' \left( \frac{n}{2} - |E| \right)$$

$$= 0 - \epsilon |S^*||\bar{S}^*| + \lambda' \left( \frac{n}{2} - |E| \right) + \epsilon \left( \frac{n}{2} - |E| \right)$$

$$= \lambda^* \left( \frac{n}{2} - |E| \right) + \epsilon \left( \frac{n}{2} - |E| - |S^*||\bar{S}^*| \right)$$

$$\geq \lambda^* \left( \frac{n(n - 1)}{2} - \frac{nd}{2} \right) + \epsilon \left( \frac{n(n - 1)}{2} - \frac{nd}{2} - \frac{n^2}{4} \right)$$

$$\geq \frac{c}{n} \left( \frac{n(n - 1)}{2} - \frac{nd}{2} \right) = \Omega(n),$$

relying on the definition of $\lambda^*$, the fact that $|E| = nd/2$ in this expander graph, and the bound $|S^*||\bar{S}^*| \leq n^2/4$.

Upper Bounding the NEPPC LP

We now show that a carefully crafted feasible solution for the NEPPC LP (4) has score $O(n/\log n)$. Let $\text{dist}(i, j)$ denote the minimum path length between nodes $i$ and $j$ in $G$, based on unit-weight edges $E$. We are assuming the graph is connected, so each $\text{dist}(i, j)$ is a finite integer. (If the graph is not connected, we ought to solve LambdaCC on each connected component separately.) Consider the following setting of values $x_{ij}$:

$$x_{ij} = \begin{cases} 2/(\log_d n) & \text{if } (i, j) \in E \\ 1 & \text{if } (i, j) \notin E \text{ and } \text{dist}(i, j) \geq (\log_d n)/2 \\ 0 & \text{if } (i, j) \notin E \text{ and } \text{dist}(i, j) < (\log_d n)/2. \end{cases}$$

We show that this is feasible for the NEPPC LP (4). Since all (positive) edges are assigned the same LP score, the NEPPC constraints are satisfied at a (negative) non-edge, $(i, j)$, if and only if $x_{ij} \leq \text{dist}(i, j) \cdot 2/(\log_d n)$. When $\text{dist}(i, j)$ is less than $\log_d n)/2$, $x_{ij} = 0$, so this inequality is trivially true. When $\text{dist}(i, j)$ is at least $\log_d n)/2$, the NEPPC inequality is true because $\text{dist}(i, j) \cdot 2/(\log_d n)$ is at least 1, which is $x_{ij}$.

For constant $d$, the contribution from the (positive) edges to LP (4) is:

$$(1 - \lambda')|E|2/(\log_d n) = (1 - \lambda')(nd)/(\log_d n) = O(n/\log n).$$
From the (negative) non-edges, since the factor is $1 - x_{ij}$, we only have a non-zero contribution from the set of $(i, j) \notin E$ such that $\text{dist}(i, j) < (\log_d n)/2 = \log_d \sqrt{n}$. For each node $v \in V$, there are at most $d^\text{poly} \sqrt{n} = \sqrt{n}$ nodes within this distance; the total number of non-edges that contribute to the LP cost is therefore in $O(n \sqrt{n})$. Each has a weight $\lambda' < 2\lambda^*$, so

$$\text{LP contribution of non-edges} \leq \lambda' n \sqrt{n} \leq (2d/(n - 1)) n \sqrt{n} = O(\sqrt{n}) \leq O(n/\log n).$$

Therefore, the total LP cost corresponding to this feasible solution to NEPPC LP (4) is $O(n/\log n)$. Since the optimal LAMBDACC solution has cost $\Omega(n)$, we have shown that there exists some $\lambda < 1/2$ such that the LP relaxation (3) has an integrality gap of $O(\log n)$.

### 3.2 Parameter-Dependent Approximation Guarantees

We now describe improved approximation guarantees for ranges of $\lambda$ below 1/2, extending the analysis of our previous 3-approximation for $\lambda \geq 1/2$ [22]. This 3-approximation is obtained by solving the LP relaxation, forming a new unweighted signed graph $G'$, and then applying the pivoting procedure, which repeatedly selects a node and clusters it with its positive neighbors. The approximation guarantee comes from applying a theorem of van Zuylen and Williamson for deterministic pivoting algorithms for correlation clustering [21]. We give a full proof of the following result in the extended version of the paper [13].

> Theorem 2. Let $(x_{ij})$ be the variables from solving the LAMBDACC LP relaxation, and form a new unweighted Correlation Clustering input $G'$ by putting a positive edge between i and j, if $x_{ij} \leq 1/3$ and a negative edge otherwise. Applying a pivoting algorithm to $G'$ yields a clustering that is a 3-approximation for $\lambda > 1/2$, and an $\alpha$-approximation otherwise, where $\alpha = \max\{1/\lambda, (6 - 3\lambda)/(1 + \lambda)\}$.

This theorem implies an approximation better than 4.5 for all $\lambda \in (0.2324, 0.5)$, but shows that the algorithm performs worse and worse as $\lambda$ decreases. However, for all $\lambda$ in $\omega(1/\log n)$, this outputs a better result than the standard, $O(\log n)$, rounding scheme.

### 3.3 Two-Cluster LambdaCC

Before moving on we note an approximation guarantee and a hardness result that holds for the two-cluster variant of LAMBDACC.

> Theorem 3. Two-cluster LAMBDACC can be reduced to the weighted MIN UNCUT problem. An instance of MIN UNCUT with non-zero optimum can be reduced to an instance of two-cluster LAMBDACC whose objective score for any clustering differs by at most a small constant factor.

We give a full proof in the full version [13]. The first fact implies the $O(\sqrt{\log n})$ approximation, due to Agarwal et al. [1], extends to 2-LAMBDACC. This has important ramifications even without the restriction on the number of clusters; LAMBDACC is guaranteed to form two clusters for a certain parameter regime near $\lambda^*$ [22, Theorem 3.2]. The reduction from MIN UNCUT to two-cluster LAMBDACC implies the latter cannot be approximated to within any constant factor [15, 14].

### 4 Motif Correlation Clustering

We now turn to improved approximations for MOTIFCC. We begin by presenting a $4(k - 1)$ approximation algorithm for the problem for hyperedges of degree $k$ with edge weights satisfying probability constraints. We then consider a first step towards algorithms that do not rely on solving an expensive LP relaxation, by showing how to obtain a combinatorial approximation for two-cluster MOTIFCC (2-MOTIFCC) for complete, unweighted instances.
44:8 Correlation Clustering Generalized

Algorithm 1 Generalized CGW for Minimizing Hyper-Disagreements.

**Input:** Signed hypergraph $G = (V, E_k)$, and threshold parameters $\gamma$ and $\delta$

Solve the LP-relaxation of ILP (2), obtaining *distances* $(x_{ij})$

$W \leftarrow V, \mathcal{C} \leftarrow \emptyset$

while $W \neq \emptyset$ do

5: Choose $u \in W$ arbitrarily, and define $T_u \leftarrow \{i \in W \setminus \{u\} : x_{ui} \leq \gamma\}$

if $\sum_{i \in T_u} x_{ui} < \gamma|T_u|$ then $S := \{u\} \cup T_u$

else $S := \{u\}$

$\mathcal{C} \leftarrow \mathcal{C} \cup \{S\}, W \leftarrow W \setminus S$

4.1 The $4(k-1)$ approximation

Our algorithm for MotifCC is closely related to the approach of Li et al. [17] and directly generalizes the LP-rounding technique of Charikar et al. [5], which is itself an instantiation of the more general rounding procedure given in Algorithm 1. The general algorithm forms clusters based on threshold parameters $\gamma$ and $\delta$, which are part of the input. Charikar et al. proved that for the $k = 2$ unweighted case of MotifCC, setting $\gamma = \delta = 1/2$ leads to a 4-approximation. Li et al. generalized this to obtain a 9-approximation for $k = 3$ in the more general probability constrained case, by selecting $\gamma = \delta = 1/3$ [17]. Although they did not provide an analysis for motifs of size $k > 3$, it appears that their strategy of setting $\gamma = \delta = 1/k$ would at best lead to a $k^2$ approximation. In contrast, we analyze a choice of parameters which leads to an approximation that is *linear* in $k$.

The result is somewhat detailed, and we begin with some notation. Let the family of $k$-tuples be $E_k$, and let $W \subseteq V$ be the subset of nodes in $G$ that remain unclustered after a certain number of rounds of Algorithm 1. When considering a vertex $u \in W$ and a specific $k$-tuple $E$, it will be convenient to define $a$ to be the node in $E$ closest to $u$, i.e., $\arg \min_{i \in E} x_{ui}$, while $z$ is the *farthest*, $\arg \max_{i \in E} x_{ui}$. We have $T_u$ similar to Algorithm 1, with $\gamma = 1/(2(k - 1))$, while $T_u^{k}$ are those $k$-tuples that include $u$, with all non-$u$ nodes in $T_u$:

$$T_u = \left\{ i \in W \setminus \{u\} : x_{ui} \leq \frac{1}{2(k-1)} \right\} \text{ and } T_u^{k} = \left\{ E \in E_k : u \in E \text{ and } (E - \{u\}) \subseteq T_u \right\}. \quad (5)$$

For $z \notin T_u$, we let $P_z$ be those $k$-tuples in which $z$ is the farthest element from $u$ and some $a \in T_u$ is closest, viz.

$$P_z = \{(a, j_2, j_3, \ldots, j_{k-1}, z) \in E_k : a \in T, x_{ua} \leq x_{u,j_2} \leq x_{u,j_3} \leq \cdots \leq x_{uz}\}. \quad (6)$$

Finally, LP($A$) denotes the LP score associated with a subset $A$ of the set of degree-$k$ hyperedges: $A \subseteq E_k$.

**Theorem 4.** For constant $k$, let $G = (V, E_k)$ be a hypergraph in which for all $E \in E_k$ the weights satisfy probability constraints, $w_E + w_{\bar{E}} = 1$. Applying Algorithm 1 with $\gamma = 1/(2(k - 1))$ and $\delta = 1/2$ outputs a clustering that is a $4(k-1)$-approximation to MotifCC.

We start with a proof outline, establish three lemmata, and then give full details in Section 4.2. At each step the algorithm forms a cluster $S_u$ around an arbitrary $u \in W$. This cluster is associated with a set of hyperedges $A_u$ that have either been cut or placed inside of $S_u$. If for each $S_u$ individually we can show that mistakes made at $A_u$ are within a fixed factor of the lower bound $LP(A_u)$, this will imply an overall bound for the entire clustering.
In forming a cluster around $u$, the algorithm first identifies a set of nodes $T_u$ whose LP distance to $u$ is at most a preliminary threshold $\gamma = 1/(2(k-1))$. To verify if $\{u\} \cup T_u$ will make a good cluster, the algorithm checks whether on average the distance from $u$ to $T_u$ is below a tighter threshold $\gamma_\delta = 1/(4(k-1))$. If this doesn’t hold, we let $\{u\}$ remain a singleton cluster. In forming clusters, we only explicitly consider distance variables $x_{ij}$ for $(i, j) \in V \times V$. However, the MotifCC objective and its LP relaxation both depend on the hyperedge variables $x_E$ for $E \in E_k$. Therefore, in order to bound the weight of hyperedge mistakes we must leverage the LP constraints to understand the relationships between distance and hyperedge variables. Lemma 5 establishes several useful relationships on the hyperedge variables $(x)$ and the weight of both positive and negative mistakes made at non-singleton clusters.

Proof. We must account for the weight of positive mistakes made at singleton clusters, $\{u\}$, and the weight of both positive and negative mistakes made at non-singleton clusters.

Singleton Clusters

Consider a cluster $S = \{u\}$. The algorithm incurs a penalty $w^+_{E}$ for each $E$ such that $u \in E$. If some node $j \in E - \{u\}$ is not in $T_u$, then the contribution to the LP score is $w^+_{E} x_E$, which is at least $w^+_{E} x_{uj}$, and therefore exceeds $w^+_{E} / (2(k-1))$. Thus the cost of the mistake at most $2(k-1)$ times the LP penalty.

It remains to account for all positive hyperedges in $T^+_u$. Even if $w^+_{E} = 1$ for all $E \in T^+_u$, $|T^+_u| = \binom{|T_u|}{k}$ is an upper bound on the total weight of mistakes made on hyperedges in $T^+_u$. By the first observation of Lemma 5, and because $u \in E$,

$$x_E \leq \sum_{i \in E} x_{ui} \leq (k-1) \frac{1}{2(k-1)} = \frac{1}{2}, \quad \text{hence,} \quad (1 - x_E) \geq x_E.$$  

Since $w^+_E + w^-_E = 1$, we can lower bound the contribution of $T^+_u$ to the LP score:

$$\text{LP}(T^+_u) = \sum_{E \in T^+_u} w^+_E x_E + w^-_E (1 - x_E) \geq \sum_{E \in T^+_u} w^+_E x_E + w^-_E x_E = \sum_{E \in T^+_u} x_E \geq |T_u| \frac{1}{4(k-1)},$$

by Lemma 6, so we have paid for the mistakes within a factor $4(k-1)$. 

\textbf{Lemma 5.} For all $E \in E_k$ and any $u \in V$, 
1. $x_E \leq \sum_{i \in E} x_{ui}$,  
2. $x_E \leq x_{ua} + (k-1)x_{uz}$, and  
3. $x_E \geq x_{uz} - x_{ua}$.

\textbf{Lemma 6.} For all $u \in W \subseteq V$, if $\sum_{i \in T_u} x_{ui} \geq \beta |T_u|$, then $\sum_{E \in T^+_u} x_E \geq \beta |T^+_u|$.

\textbf{Lemma 7.} For all $E \in P_z$, let $a_E$ denote the node in $E$ closest to $u$. If $\sum_{i \in T_u} x_{ui} < \beta |T_u|$, then $\sum_{E \in P_z} x_{uaE} < \beta |P_z|$.

4.2 Proof of Theorem 4

Proof. We must account for the weight of positive mistakes made at singleton clusters, $\{u\}$, and the weight of both positive and negative mistakes made at non-singleton clusters.
Negative Mistakes at Non-Singletons

Next, we account for negative mistakes in clusters of the form $S = \{u\} \cup T$. Charikar et al. showed that, when $k = 2$, these are accounted for within a factor 4; we prove the same for all $k \geq 3$. For each $E \in E_k$ such that $E \subseteq S$, the algorithm makes a mistake of weight $w_E^-$. On the other hand, the LP pays $w_E^-(1 - x_E)$. Applying the first observation in Lemma 5,

$$x_E \leq \sum_{u \in E} x_{uu} \leq k \frac{1}{2k-1} \leq \frac{3}{4},$$

hence, $w_E^-(1 - x_E) \geq \frac{w_E^-}{4}$, and we have the desired result for $k \geq 3$.

Positive Mistakes at Non-Singletons

A hyperedge $E$ contained entirely within $S = \{u\} \cup T$ incurs no positive-weight error. So, finally, we account for positive mistakes at hyperedges $E$ where at least one node of $E$ is in $S$ and at least one node in $E$ is $\notin S$. For each such hyperedge, we explicitly label the nodes of $E$ with indices $a = j_1 < j_2 < \cdots < j_k = z$, with $x_{ua} = x_{u,j_1} \leq x_{u,j_2} \leq \cdots \leq x_{u,j_k} = x_{uz}$ where $a \in T_u$ and $z \notin T_u$. By the second and third observation in Lemma 5 we know that

$$x_{uza} - x_{ua} \leq x_E \leq x_{ua} + (k-1)x_{uz}, \quad (7)$$

First, if $a = u$, then we know $w_E^- x_E \geq w_E^+(x_{uza} - x_{ua}) > w_E^-/(2(k-1))$, and we have individually accounted for each such positive mistake within a factor $2(k-1)$. If $a \neq u$ and $x_{uza} \geq 3/(4(k-1))$, we bound the mistake within factor $4/(k-1)$:

$$w_E^- x_E \geq w_E^+(x_{uza} - x_{ua}) \geq w_E^- (3/(4(k-1)) - 1/(2(k-1))) = w_E^-/(4(k-1)).$$

Finally, if $a \neq u$ and $x_{uza} \leq (1 - 3/(4k-1))$, we account for all positive weights associated with edges in the following set, together:

$$P_z = \{E \in E_k : E = \{a, j_2, \ldots, z\}, a \in T, x_{ua} \leq x_{u,j_2} \leq x_{u,j_3} \leq \cdots \leq x_{uza}\}.$$  

The weight of mistakes made by the algorithm is $W_z^+ = \sum_{P \in P_z} w_p^+$, and we also define $W_z^- = \sum_{P \in P_z} w_p^-$. We start by observing that, since $x_{ua} \leq x_E$ and $W_z^+ + W_z^- = |P_z|$, due to probability constraints on weights, Lemma 7 tells us that $\sum_{P \in P_z} x_{ua} \leq (W_z^+ + W_z^-)/(4(k-1))$.

$$LP(P_z) = \sum_{E \in P_z} w_E^- x_E + w_E^-(1 - x_E) \geq \sum_{E \in P_z} w_E^+(x_{uza} - x_{ua}) + w_E^- (1 - x_{ua} - (k-1)x_{uz}) \quad \text{(by inequalities in (7))}$$

$$= \sum_{E \in P_z} w_E^+ x_{uza} + w_E^- (1 - (k-1)x_{uz}) - \sum_{E \in P_z} x_{ua} \geq W_z^+ x_{uza} + W_z^- (1 - (k-1)x_{uz}) - \frac{W_z^+ + W_z^-}{4(k-1)} \quad \text{(by the starting observation)}$$

$$\geq W_z^+ \left(\frac{1}{2k-1} - \frac{1}{4k-1}\right) + W_z^- \left(1 - \frac{1}{4k-1} - (k-1)\frac{3}{4k-1}\right) \geq \frac{1}{4k-1},$$

so the mistakes on all hyperedges in $P_z$ are, collectively, accounted for within factor $1/(4(k-1))$, concluding the Proof of Theorem 4.

We outline two immediate extensions of this theorem in the full version [13]. First we note that the same approximation guarantees holds for the MIXED MOTIF CORRELATION CLUSTERING objective, considered by Li et al. We then consider a hybrid LAMBDA-MCC objective in which positive hyperedges have weight $(1 - \lambda)$ and negative hyperedges have weight $\lambda$, for which the algorithm is guaranteed to produce the same approximation factor when $\lambda \geq 1/2$. ❄
Algorithm 2 Pick-A-Pivot-Tuple.

**Input:** An instance of 2-MotifCC: $G = (V, E_k)$ be a hypergraph where $(w^+_E, w^-_E) \in \{(0,1), (1,0)\}$ for every $k$-tuple.

for $(k-1)$-tuple $\mathcal{K} \subseteq V$ do

$\mathcal{C}_\mathcal{K} \leftarrow$ the clustering formed by placing $\mathcal{K}$ in a cluster with all $u$ such that $\mathcal{E} = \mathcal{K} \cup \{u\}$ is positive, and placing all remaining nodes in the other cluster.

Return the $\mathcal{C}_\mathcal{K}$ with fewest mistakes.

4.3 Two-Cluster MotifCC

The LP relaxation of MotifCC involves $O(n^k)$ variables and $O(n^k)$ constraints for all $k > 2$, and is therefore very expensive to solve in practice. For standard Correlation Clustering, only a few of the known approximation algorithms avoid solving an expensive convex relaxation [2, 3]; it is natural to ask whether a similar, combinatorial, approach can be taken for MotifCC. We give first steps in this direction, with a constant-factor combinatorial approximation algorithm for MotifCC, when the output is restricted to two clusters, generalizing the 3-approximation of Bansal et al. for 2-Correlation Clustering [3]. Our method is shown in Algorithm 2. We call this algorithm Pick-a-Pivot-Tuple, and show it satisfies the following result:

**Theorem 8.** For a constant integer $k > 1$, Algorithm 2 returns a $(1 + kc)$-approximation for 2-MotifCC, where $c \leq 2^{k-2}$ for $k = 2, 3$, while $\lim_{n \to \infty} c = 2^{k-2}$ for $k > 3$.

We give a proof of the above result in the full version [13]. Although the exponential dependence on $k$ makes this a poor approximation for large motifs, at least in the case $k = 3$, this is a 7-approximation for all $n$, not just for large $n$.

5 Discussion

We have demonstrated a $\Theta(\log n)$ integrality gap for the LambdaCC LP relaxation, which highlights why previous attempts to obtain a constant-factor approximation via LP rounding have failed. It remains an open question whether better approximation factors exist for small values of $\lambda$ in $O(1/\log n)$. For minimizing disagreements, there are relatively few techniques that don’t rely on the LP relaxation that lead to approximations better than $O(\log n)$ for different variants of correlation clustering. The next step is either to develop an entirely new approach or prove further hardness results for approximating LambdaCC when $\lambda$ is small.

For MotifCC, we have given an approximation algorithm for arbitrary (constant) hyperedge size $k$ that is linear in $k$, and provided a first combinatorial approximation result, which avoids solving an LP relaxation, for to the two-cluster case. An interesting open question is whether a pivoting algorithm à la Ailon et al. [2] could be developed for the MotifCC objective. For maximizing agreements, the simple strategy of either placing all nodes together or separating all nodes into singletons will still lead to a 1/2-approximation for hypergraphs with arbitrary weights and any $k$. This leads to open questions about what results for maximizing agreements can be generalized to the hypergraph setting. Another open question is whether an approximation that is independent of $k$ could be developed for minimizing disagreements in hypergraphs.
References


Partitioning Vectors into Quadruples: Worst-Case Analysis of a Matching-Based Algorithm

Annette M. C. Ficker
Faculty of Economics and Business, KU Leuven, Leuven, Belgium
Annette.Ficker@3ds.com

Thomas Erlebach
Department of Informatics, University of Leicester, Leicester, United Kingdom
t.erdebach@leicester.ac.uk
https://orcid.org/0000-0002-4470-5868

Matúš Mihalák
Department of Data Science and Knowledge Engineering, Maastricht University, Maastricht, The Netherlands
matus.mihalak@maastrichtuniversity.nl
https://orcid.org/0000-0002-1898-607X

Frits C. R. Spieksma
Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands
f.c.r.spieksma@tue.nl
https://orcid.org/0000-0002-2547-3782

Abstract

Consider a problem where $4k$ given vectors need to be partitioned into $k$ clusters of four vectors each. A cluster of four vectors is called a quad, and the cost of a quad is the sum of the component-wise maxima of the four vectors in the quad. The problem is to partition the given $4k$ vectors into $k$ quads with minimum total cost. We analyze a straightforward matching-based algorithm and prove that this algorithm is a $\frac{3}{2}$-approximation algorithm for this problem. We further analyze the performance of this algorithm on a hierarchy of special cases of the problem and prove that, in one particular case, the algorithm is a $\frac{5}{4}$-approximation algorithm. Our analysis is tight in all cases except one.

2012 ACM Subject Classification Mathematics of computing → Approximation algorithms

Keywords and phrases approximation algorithm, matching, clustering problem

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.45

Related Version A full version of the paper is available at [6], https://arxiv.org/abs/1807.01962.
1 Introduction

Partitioning Vectors into Quadruples (PQ) is the problem of partitioning $4k$ given nonnegative vectors $v_1, \ldots, v_{4k}$, each consisting of $n$ components, into $k$ clusters, each containing exactly four vectors. We refer to such a cluster of four vectors as a *quad* or a *quadruple* for short. The cost of a quad $Q = \{v_{i1}, v_{i2}, v_{i3}, v_{i4}\}$ is the sum of the component-wise maxima of the four vectors in the quad. The goal of the problem is to find a partition of the $4k$ vectors into $k$ quads such that the total cost of all quads is minimum.

We will analyze the following matching-based algorithm, called algorithm $A$, that finds a solution to problem PQ by proceeding in two phases. In the first phase, algorithm $A$ builds a complete, edge-weighted graph $G = (V, E)$ that has a node in $V$ for each vector in the instance (hence $|V| = 4k$). The weight of an edge equals the sum of the component-wise maxima of the two vectors whose corresponding nodes span the edge. Now, algorithm $A$ computes a minimum-cost perfect matching $M$ in the complete graph $G$, yielding $2k$ vector pairs. Let $p_1, \ldots, p_{2k}$ be the $2k$ matched vector pairs corresponding to the computed matching $M$. In the second phase, algorithm $A$ builds a complete, edge-weighted graph $G' = (V', E')$ that has a node in $V'$ for each vector pair $p_i$ found in the first phase ($i = 1, \ldots, 2k$; $|V'| = 2k$). The weight of an edge equals the sum of the component-wise maxima of the two vector pairs whose corresponding nodes span the edge. Now, algorithm $A$ computes a minimum-cost perfect matching $M'$ in the complete graph $G'$. Each of the $k$ edges of $M'$ matches two vector pairs, which naturally induces a quad. The $k$ quads induced by the edges of $M'$ constitute a solution to the problem. Clearly, $A$ is a polynomial-time algorithm. A rigorous description can be found in Section 2. It is not hard to see that algorithm $A$ may fail to find an optimum solution for an instance of the problem, i.e., $A$ is not exact, and we are interested in analyzing how far off algorithm $A$’s output can be from an optimum solution.

In this paper we show that $A$ is a $3/2$-approximation algorithm for problem PQ, and that this bound is tight. We also show that algorithm $A$ has better approximation guarantees for various special cases of problem PQ. In particular, consider an instance of PQ where each vector has exactly two ones, while all other components are zero. In that case, each vector can be seen as an edge in a graph where there is a node for each component. For the case where this graph is a simple, connected graph, we prove that $A$ is a $5/4$-approximation algorithm. We give a precise overview of our results in Section 2.3.

The paper is organized as follows. The remainder of this section introduces some terminology and discusses related work that motivates our research. Section 2 gives preliminaries and states our results. In Section 3, we give the proof of the upper bound on the worst-case ratio of algorithm $A$ for the special case of problem PQ mentioned above; we also outline the proofs for all other cases. Section 4 contains the lower bound result for the special case. We conclude in Section 5. Detailed proofs of all our bounds on the worst-case ratio of algorithm $A$ for problem PQ and other generalizations of the special case can be found in [6].

1.1 Terminology and related literature

Worst-case analysis is a well-established tool to analyze the quality of solutions found by heuristics. We refer to books by Vazirani [13] and Williamson and Shmoys [14] for a thorough introduction to the field. We use the following, standard terminology that applies to minimization problems. In the next definition, $A(I)$ stands for the value of the solution to instance $I$ found by algorithm $A$, while $OPT(I)$ stands for the value of an optimum solution to instance $I$.
Definition 1. Algorithm $A$ is an $\alpha$-approximation algorithm for a minimization problem $P$ if for every instance $I$ of problem $P$: (i) algorithm $A$ runs in polynomial time, and (ii) $A(I) \leq \alpha \cdot OPT(I)$. We refer to $\alpha$ as an upper bound on the worst-case ratio of algorithm $A$.

Different problems in various fields are related to problem $PQ$, and share some of its characteristics. In addition, algorithm $A$ can often be adjusted to work in a particular setting. We now review related literature and provide a number of such examples.

Onn and Schulman [10] consider a problem where a given set of vectors in $n$-dimensional space needs to be partitioned into a given number of clusters. The number of vectors in a cluster (its size) is not specified, and in addition, they assume that the objective function, which is to be maximized, is convex in the sum of the vectors in the same cluster. Their framework contains many different problems with diverse applications, and they show, for their setting, strongly-polynomial time, exact algorithms. This is in contrast to our problem which is NP-hard (cf. Section 2.1).

Another problem, distinct from, yet related to, our problem, comes from computational biology, and is described in Figuero et al. [7]. Here, a component of a vector is a 0 or a 1 or an “N”. In this setting neither the size of a cluster, nor the number of clusters is fixed; the goal is to find a partition of the set of vectors into a minimum number of clusters while satisfying the condition that a pair of vectors that is in the same cluster can only differ at a component where at least one of them has the value N. They prove hardness of this problem, and analyze the approximation behavior of heuristics for this problem.

Hochbaum and Levin [8] describe a problem in the design of optical networks that is related to our special case where each vector is a $\{0, 1\}$-vector containing two ones. In essence, their problem is to cover the edges of a given bipartite graph by a minimum number of 4-cycles. They observe that this problem is a special case of unweighted 4-set cover; they give a $\left(\frac{13}{10} + \epsilon\right)$-approximation algorithm (using local search), and analyze the performance of a greedy algorithm for a more general version of the problem. Our problem differs from theirs in the sense that we deal with a partitioning problem, where there is a weight for each set; in addition, our problem does not necessarily have a bipartite structure, nor do our quads need to correspond to 4-cycles.

Our problem is also intimately related to a problem occurring in wafer-to-wafer yield optimization (see, e.g., Reda et al. [11] for a description). Central in this application is the production of so-called waferstacks, which can be seen as a set of superimposed wafers. In our context, a wafer can be represented by a vector. A wafer consists of many dies, each of which can be in two states: either functioning, i.e., good (which corresponds to a component in the vector with value ‘0’), or malfunctioning, i.e., bad (which corresponds to a component in the vector with value ‘1’). The quality of a waferstack is measured by simply counting the number of components that have only 0’s in the wafers contained in the waferstack. The goal is to partition the set of wafers into waferstacks (clusters) such that total quality is as high as possible. In this application, however, there are different types of wafers, and a waferstack needs to consist of one wafer of each type. This would correspond to an a priori given partition of the vectors. In addition, a typical waferstack consists in practice of many, i.e., more than 4, wafers. Dokka et al. [4] analyze the worst-case behavior of different algorithms that have as a common feature solving assignment problems repeatedly. The case where there are three types of wafers, and the problem is to find waferstacks that are triples containing one wafer of each type, is investigated in Dokka et al. [3]; for a particular objective function, they describe a $\frac{4}{3}$-approximation algorithm.

A restricted, yet very relevant special case of our problem is one where the edges of a given graph need to be partitioned into subsets each containing four edges (see Section 2 for a precise description). Indeed, from a graph-theoretical perspective, there is quite some interest
and literature in partitioning the edge-set of a graph, i.e., to find an edge-decomposition. In fact, edge-decompositions where each cluster has prescribed size have already been studied in e.g. Jünger et al. [9], Thomassen [12] studies the existence of edge-decompositions into paths of length 4, and Barat and Gerbner [1] even study edge-decompositions where each cluster is isomorphic to a tree consisting of 4 edges.

\section{Preliminaries}

\subsection{About problem PQ: special cases and complexity}

We first observe that, for the analysis of algorithm $A$, we can restrict ourselves to instances of problem PQ where the $4k$ vectors are \{0,1\}-vectors. Notice that we call a vector nonnegative when each of its entries is nonnegative.

\begin{lemma}
Each instance of problem PQ with arbitrary (rational) nonnegative vectors can be reduced to an instance of problem PQ with \{0,1\}-vectors.
\end{lemma}

The argument in the proof of Lemma 2 (see [6]) implies that any worst-case ratio of algorithm $A$ shown to hold for instances consisting of \{0,1\}-vectors holds in fact for arbitrary rational nonnegative vectors. Clearly, this does not mean that algorithm $A$ is restricted to work on instances consisting of binary vectors; it works directly on the original input vectors.

Thus, from hereon we restrict ourselves, without loss of generality, to the case of binary vectors. There are various special cases of PQ that are of independent interest. We will describe the particular special case in brackets following ‘PQ’; we distinguish the following special cases.

\begin{itemize}
\item Problem PQ($\#1 \in \{1,2\}$). The case where each vector contains either one or two 1’s; all other components have value 0. It will turn out that, at least in terms of the worst-case behavior of algorithm $A$, this special case displays the same behavior as the general problem PQ.
\item Problem PQ($\#1 = 2$). The case where each binary vector contains exactly two 1’s. Instances of this type can be represented by a multi-graph $F$ with $n$ nodes, each node corresponding to a component of a vector. Each vector is then represented by an edge spanning the two nodes that correspond to components with value 1. Of course, now a quad can be seen as a set of four edges, and its cost equals the number of nodes in the subgraph induced by these four edges.
\item Problem PQ($\#1 = 2$, distinct). The case where the graph $F$ is a simple graph. Equivalently, this means that each vector contains exactly two 1’s and the vectors are pairwise distinct.
\item Problem PQ($\#1 = 2$, distinct, connected). We distinguish a further special case by demanding that the graph $F$ is also connected.
\end{itemize}

Clearly, the special cases are ordered, in the sense that each next one is a special case of its predecessor.

Although our interest is on the worst-case behavior of algorithm $A$, it is relevant to establish the computational complexity of problem PQ. It turns out (see [6] for the proof) that even its special case PQ($\#1 = 2$, distinct, connected) is NP-hard. This fact shows that no polynomial-time algorithm for problem PQ can be exact, unless P=NP.

\begin{theorem}
PQ($\#1 = 2$, distinct, connected) is NP-hard.
\end{theorem}
2.2 About algorithm A: notation and properties

Recall that, in our analysis, we may assume that all vectors are \{0,1\}-vectors. Let \( v_i \lor v_j \) denote the vector that is the component-wise maximum of the two vectors \( v_i \) and \( v_j \), i.e.:

\[
v_i \lor v_j = (\max(v_{i,1},v_{j,1}), \max(v_{i,2},v_{j,2}), \ldots, \max(v_{i,n},v_{j,n})).
\]

Here, \( v_{i,\ell} \) denotes the \( \ell \)-th component of vector \( v_i \) \( (\ell = 1, \ldots, n) \). We use \( |v_i| \) to denote the number of ones in vector \( v_i \) \( (1 \leq i \leq 4k) \), i.e., \(|v_i| = \sum_{\ell=1}^n v_{i,\ell} \). The cost of a quad \( Q = \{v_1,v_2,v_3,v_4\} \) is then \( \text{cost}(Q) = |v_1 \lor v_2 \lor v_3 \lor v_4| \). For a pair \( p = \{v_1,v_2\} \) of vectors, we set \( \text{cost}(p) = |v_1 \lor v_2| \).

For two vectors \( v_i \) and \( v_j \), let \( \text{sav}(v_i,v_j) \) (the “savings” made by combining \( v_i \) and \( v_j \)) denote the number of common ones in \( v_i \) and \( v_j \), i.e.:

\[
\text{sav}(v_i,v_j) = \sum_{\ell=1}^n \min(v_{i,\ell},v_{j,\ell}).
\]

If \( p = \{v_1,v_2\} \) and \( p' = \{v_3,v_4\} \) are pairs of vectors, we also write \( \text{sav}(p,p') \) for \( \text{sav}(v_1 \lor v_2, v_3 \lor v_4) \).

The following observation concerning two \{0,1\}-vectors \( u \) and \( v \) is immediate.

**Observation 4.** \(|u| + |v| = \text{sav}(u,v) + |u \lor v|\).

Let us revisit the description of Algorithm A. In the first phase, it computes a minimum-cost perfect matching \( M \) in the complete graph \( G \) on the given \( 4k \) vectors, where the weight of the edge between vectors \( v_i \) and \( v_j \) is set to \(|v_i \lor v_j|\). Let \( p_1, \ldots, p_{2k} \) be the \( 2k \) matched vector pairs corresponding to the computed matching \( M \), and let \( \text{cost}(M) \) denote the cost of the matching \( M \). For \( 1 \leq i \leq 2k \), let \( v_i^1 \) and \( v_i^2 \) be the two vectors in the vector pair \( p_i \), and let \( v_i^1 = v_i^1 \lor v_i^2 \).

In the second phase, Algorithm A computes a minimum-cost perfect matching \( M' \) in the complete graph \( G' \) on the \( 2k \) vector pairs, where the weight of the edge between pairs \( p_i \) and \( p_j \) is set to \(|v_i^1 \lor v_j^1|\). The quads corresponding to \( M' \) are output as a solution. Let \( \text{cost}(M') \) be the cost of matching \( M' \).

**Observation 5.** \( \text{A}(I) = \text{cost}(M') \) and \( \text{cost}(M') \leq \text{cost}(M) \).

**Lemma 6.** In the first phase of algorithm A, we can equivalently set the weight of the edge between \( v_i \) and \( v_j \) to be \( -\text{sav}(v_i,v_j) \). Similarly, in the second phase of algorithm A, we can set the weight of the edge between \( p_i \) and \( p_j \) to be \( -\text{sav}(v_i^1,v_j^1) \).

Let \( \text{weight}(M') \) denote the total savings of the perfect matching \( M' \), i.e., \( \text{weight}(M') = \sum_{(v_i^1,v_j^1) \in M'} \text{sav}(v_i^1,v_j^1) \). Then, we have:

\[
\text{cost}(M') = \text{cost}(M) - \sum_{(v_i^1,v_j^1) \in M'} \text{sav}(v_i^1,v_j^1) = \text{cost}(M) - \text{weight}(M'). \tag{1}
\]

Observation 5 and Equation (1) imply:

**Corollary 7.** \( \text{A}(I) = \text{cost}(M) - \text{weight}(M') \).

In view of this corollary, it follows that if we can show that \( \text{cost}(M) \leq B \) and \( \text{weight}(M') \geq S \) for some bounds \( B \) and \( S \), we can conclude that \( \text{A}(I) \leq B - S \).

Two vectors \( u \) and \( v \) are identical when \( u = v \), and a pair of identical vectors is called an identical pair. In the following we show that among the set of minimum-cost perfect matchings, there is one that contains a maximum number of identical pairs.
Table 1 Overview of bounds on the worst-case ratio of algorithm $A$. Proofs of the bounds marked with (*) are omitted due to space restrictions and can be found in [6].

<table>
<thead>
<tr>
<th>Problem name</th>
<th>Lower Bound</th>
<th>Upper Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>$PQ$</td>
<td>$\frac{3}{2}$</td>
<td>$\frac{3}{2}$ (*)</td>
</tr>
<tr>
<td>$PQ(#1 \in {1, 2})$</td>
<td>$\frac{3}{2}$ (*)</td>
<td>$\frac{3}{2}$ (*)</td>
</tr>
<tr>
<td>$PQ(#1 = 2)$</td>
<td>$\frac{4}{3}$ (*)</td>
<td>$\frac{4}{3}$ (*)</td>
</tr>
<tr>
<td>$PQ(#1 = 2, \text{distinct})$</td>
<td>$\frac{4}{3}$ (Observation 16)</td>
<td>$\frac{5}{4}$ (Lemma 14)</td>
</tr>
</tbody>
</table>

Lemma 8. There is a minimum-cost perfect matching in $G$, as well as in $G'$, that contains a maximum number of identical pairs.

Thus, in the implementation of our algorithm $A$, we can first greedily match pairs of identical vectors as long as they exist, and then use any standard minimum-cost perfect matching algorithm to compute a perfect matching of the remaining vectors.

2.3 Our results

In this paper, we show the following bounds on the worst-case ratio of algorithm $A$ (see Table 1 for a summary).

- **Theorem 9.** Algorithm $A$ is a $\frac{3}{2}$-approximation algorithm for problem $PQ$, and this bound is tight.
- **Theorem 10.** Algorithm $A$ is a $\frac{3}{2}$-approximation algorithm for problem $PQ(#1 \in \{1, 2\})$, and this bound is tight.
- **Theorem 11.** Algorithm $A$ is a $\frac{4}{3}$-approximation algorithm for problem $PQ(#1 = 2)$, and this bound is tight.
- **Theorem 12.** Algorithm $A$ is a $\frac{13}{10}$-approximation algorithm for problem $PQ(#1 = 2, \text{distinct})$, and its worst-case ratio is at least $\frac{5}{4}$.
- **Theorem 13.** Algorithm $A$ is a $\frac{5}{4}$-approximation algorithm for problem $PQ(#1 = 2, \text{distinct, connected})$, and this bound is tight.

The proof of Theorem 13 is given in the next sections: the proof implying the upper bound (Lemma 14) is in Section 3.1, and the instance leading to the lower bound result (Observation 16) is in Section 4.1. In Section 3.2 we provide a high-level description of the proofs leading to the other upper bound results. Full proofs of all upper and lower bounds can be found in [6].

As an aside, we also give instances that show that the worst-case ratio of a natural greedy algorithm is worse than the worst-case ratio of algorithm $A$, both for problem $PQ(#1 = 2, \text{distinct, connected})$ (Section 4.2) and for problem $PQ$ (see [6]).

3 Upper bound proofs

In this section, we prove that $\frac{5}{4}$ is an upper bound for the worst-case ratio of algorithm $A$ for Problem $PQ(#1 = 2, \text{distinct, connected})$. The proofs for the upper bound $\frac{3}{2}$ for the worst-case ratio of Problem $PQ$, the upper bound $\frac{4}{3}$ for the worst-case ratio of Problem $PQ(#1 = 2)$, and the upper bound $\frac{13}{10}$ for the worst-case ratio of Problem $PQ(#1 = 2, \text{distinct})$ can be found in [6]. An outline of our approach to derive these results is given in Section 3.2.
3.1 Approximation analysis for \( \text{PQ}(\#1 = 2, \text{distinct, connected}) \)

**Lemma 14.** Algorithm \( A \) is a \( \frac{5}{4} \)-approximation algorithm for \( \text{PQ}(\#1 = 2, \text{distinct, connected}) \).

**Proof.** Recall that an instance of \( \text{PQ}(\#1 = 2, \text{distinct, connected}) \) can be viewed as a simple, connected graph \( F \) with \( 4k \) edges, and that the cost of a quad is the number of vertices spanned by the edges in the quad. Note that the cost of every optimal quad is at least 4 since \( 4 \) edges in a simple graph touch at least \( 4 \) different vertices. Hence, \( \text{OPT} \geq 4k \). Furthermore, if we can show that there are \( z \) quads in the optimal solution that have cost at least \( 5 \), we get that \( \text{OPT} \geq 4(k - z) + 5z = 4k + z \).

**Observation 15.** \( \text{cost}(M) = 6k \).

**Proof.** The line graph of a connected graph with an even number of edges admits a perfect matching (Jünger et al. [9], Dong et al. [5]). Thus, the minimum-cost perfect matching \( M \) pairs adjacent edges of the graph. Hence, every pair in \( M \) has cost 3, and thus the cost of \( M \) is \( 2k \cdot 3 = 6k \).}

Let \( p_1, \ldots, p_{2k} \) be the pairs corresponding to \( M \). Consider the auxiliary graph \( H \) with vertex set \( V' = \{p_1, \ldots, p_{2k}\} \) in which an edge is added between \( p_i \) and \( p_j \) if \( p_i \) and \( p_j \) have at least one common vertex (implying that matching \( p_i \) to \( p_j \) in the matching \( M' \) that \( A \) computes in the second phase would create a saving of at least one). Note that \( H \) is connected as \( F \) is connected. Let \( \mu \) be the size of a maximum matching in \( H \), \( 1 \leq \mu \leq k \). Note that the maximum matching of \( H \) can be extended to a perfect matching of \( V' \) that makes savings at least \( \mu \). Therefore, we have

\[
A(I) \leq 6k - \mu.
\]

If \( H \) contains a perfect matching, we have \( \mu = k \) and hence \( A(I) \leq 5k \), implying that \( A(I)/\text{OPT}(I) \leq 5k/(4k) = \frac{5}{4} \). It remains to consider the case \( \mu < k \).

If a maximum matching in \( H \) has size \( \mu < k \), the number of unmatched vertices is \( 2k - 2\mu \). We will show that the optimal solution then contains at least \( k - \mu \) quads with cost at least \( 5 \), and hence we have \( \text{OPT}(I) \geq 4k + (k - \mu) = 5k - \mu \). Therefore,

\[
\frac{A(I)}{\text{OPT}(I)} \leq \frac{6k - \mu}{5k - \mu} \leq \frac{5}{4},
\]

where the last inequality follows because \( (6k - \mu)/(5k - \mu) \) is maximized if \( \mu \) takes its maximum possible value, \( \mu = k \).

It remains to show that the optimal solution contains at least \( k - \mu \) quads with cost at least \( 5 \). Recall that a maximum matching in \( H \) leaves \( 2k - 2\mu \) vertices unmatched. By the Tutte-Berge formula [2], the number of unmatched vertices of a maximum matching in \( H \) is equal to

\[
\max_{X \subseteq V} (\text{odd}(H - X) - |X|),
\]

where \( \text{odd}(H - X) \) is the number of connected components of \( H - X \) that have an odd number of vertices (\( H - X \) is the graph that results when the nodes in \( X \), and their incident edges, are removed from \( H \)). Hence, there exists a set \( X \subseteq V' \) such that \( \text{odd}(H - X) - |X| = 2k - 2\mu \).

Let \( d = \text{odd}(H - X) \), and let \( O_1, O_2, \ldots, O_d \) denote the \( d \) odd components of \( H - X \). We have

\[
2k - 2\mu = d - |X|.
\]
For a subgraph $S$ of $H$, let $E_F(S)$ denote the set of edges of $F$ that are contained in the edge pairs that form the vertex set of $S$ (recall that the vertices of $H$ are pairs of edges from $F$). Note that $|E_F(O_i)| \mod 4 = 2$ for $1 \leq i \leq d$ as $O_i$ contains an odd number of edge pairs. Therefore, each $E_F(O_i)$ contains at least two edges that are contained in optimal quads that do not only contain edges from $E_F(O_i)$. If such a quad contains three edges from $E_F(O_i)$, note that there must be at least one other optimal quad that contains at most three edges from $E_F(O_i)$ as $(|E_F(O_i)| - 3) \mod 4 = 3$.

For each optimal quad that contains one or two edges from $E_F(O_i)$, define these one or two edges to be special edges. For each optimal quad that contains three edges from $E_F(O_i)$, select one of these three edges arbitrarily and define it to be a special edge. There are at least two special edges in each $E_F(O_i)$, $1 \leq i \leq d$, and hence at least $2d$ special edges in total. More precisely, we refer to these special edges as the edge-set $SE$, and partition it into two subsets: those special edges occurring in a quad with cost 4 (the set $SE4$), and those special edges occurring in a quad with cost at least 5 (the set $SE5$). Clearly:

$$2d \leq |SE4| + |SE5|. \quad (2)$$

Consider a quad with cost 4 from the optimum solution. It consists of four edges of $F$. Since $F$ is a connected simple graph there are only two possible subgraphs induced by $Q$, as depicted in Figure 1. These four edges can be in the sets $E_F(O_i)$ for some $1 \leq i \leq d$, the set $E_F(X)$, and the sets $E_F(C)$ for even components $C$ of $H - X$. We now define types of quads of cost 4 depending on how many edges are in which set.

Note that an edge from $E_F(O_i)$ cannot be incident to the same vertex as an edge from $E_F(O_j)$ for $j \neq i$ because otherwise $H$ would contain an edge between $O_i$ and $O_j$. Similarly, an edge from $E_F(O_i)$ cannot be incident to the same vertex as an edge from $E_F(C)$ where $C$ is an even component of $H - X$. The only edges that can share endpoints with edges in $E_F(O_i)$ are those in $E_F(X)$.

We tabulate the different types of quads with cost 4 in Table 2. Thus, a quad with cost 4 with a special edge must be of type 1, 2, 3, 4 or 5. For each of these types, the number of edges from $E_F(X)$ is at least the number of special edges in the quad. Thus,

$$|E_F(X)| \geq |SE4|. \quad (3)$$

Further, since $|E_F(X)| = 2|X|$, it follows from (3) and (2) that $|SE5| \geq 2d - 2|X|$. Thus, the number of quads of cost at least 5 is at least $\frac{2d - 2|X|}{4} = \frac{1}{2}(d - |X|) = k - \mu$. ◀

### 3.2 Outline of approximation analysis for other variants of PQ

In this section we give a high-level description of the crucial arguments we need to prove the three upper bound results for problems PQ, PQ(#1 = 2), and PQ(#1 = 2, distinct). As mentioned before, the full proofs are omitted due to space restrictions and can be found in [6].

To analyze algorithm $A$ for PQ, we proceed along the following lines. By Corollary 7, we have $A(I) = \text{cost}(M) - \text{weight}(M')$. We fix an arbitrary optimal solution and define from
it a perfect matching $\hat{M}$ in $G$ and an amount of savings, written in the form $S_1 + \frac{1}{2}S_2$ for reasons explained below, that algorithm $A$ can definitely achieve in the second phase. As $\text{cost}(\hat{M}) \leq \text{cost}(\hat{M})$ and weight($\hat{M}$) $\geq S_1 + \frac{1}{2}S_2$, we have $A(I) \leq \text{cost}(\hat{M}) - (S_1 + \frac{1}{2}S_2)$.

The existence of the savings $S_1 + \frac{1}{2}S_2$ is shown by constructing a subgraph $H$ of $G'$ that is bipartite, has maximum degree 2, and in which each edge connects two vertices of the same degree. $H$ consists of even-length cycles and isolated edges. Let $S_2$ be the total savings of the edges on cycles and $S_1$ the total savings of isolated edges in $H$. It follows that $H$ contains a matching with total savings at least $S_1 + \frac{1}{2}S_2$, and thus $G'$ contains a perfect matching with at least those savings.

The matching $\hat{M}$ and the graph $H$ are determined by considering each quad $Q$ of the optimal solution separately. For each quad $Q = \{v_1, v_2, v_3, v_4\}$ we define two vector pairs of $\hat{M}$ (by partitioning $Q$ into two vector pairs in one of the three possible ways) and add to $H$ either one edge (that becomes an isolated edge), or two edges (that will eventually be part of a cycle). For example, if the algorithm has matched $p = \{v_1, v_2\}$, $p_1 = \{v_3, v_4\}$ and $p_2 = \{v_4, v_4\}$ in $M$, where $v_4$ and $v_4$ are vectors not in $Q$, the edges added to $H$ are $(p, p_1)$ and $(p, p_2)$. As another example, if the algorithm has matched $p_i = \{v_i, v_3\}$ for $1 \leq i \leq 4$, we can show that we can add two disjoint edges of the form $(p_i, p_j)$ for $i \neq j$ to $H$ in such a way that $H$ remains bipartite, and that there are two different ways of selecting these two edges.

In this way, each quad $Q$ contributes an amount $\phi_Q$ to cost($\hat{M}$) $- (S_1 + \frac{1}{2}S_2)$ that consists of the weight of the two edges it adds to $\hat{M}$ minus the savings of the edge it adds to $H$ (if it adds only one isolated edge), or minus the savings of the two edges that it adds to $H$ divided by two (otherwise). By selecting the edges added to $\hat{M}$ and $H$ carefully among the valid possibilities, we can show that $H$ has the desired properties and $\phi_Q \leq \frac{3}{2} \text{cost}(Q)$ holds for each quad $Q$ of the optimal solution. Since cost($\hat{M}$) $- (S_1 + \frac{1}{2}S_2) = \sum_{Q} \phi_Q$, this implies $A(I) \leq \frac{3}{2} \text{OPT}(I)$, showing that $A$ is a $\frac{3}{2}$-approximation algorithm for problem PQ.

Now consider problem PQ($\#1 = 2$). Recall that the 4k input vectors can be viewed as edges in a multi-graph. Denote that multi-graph by $F$. To analyze algorithm $A$ for PQ($\#1 = 2$), we follow the same approach as for PQ, but obtain the better bound $\phi_Q \leq \frac{4}{3} \text{cost}(Q)$ for each optimal quad $Q$ by making a case distinction regarding the value of cost($Q$) and considering for each value of cost($Q$) all possible subgraphs of $F$ that the edges of $Q$ can induce. For example, if cost($Q$) $= 3$, one of the cases is that the subgraph induced by $Q$ is a 3-cycle with one duplicate edge. Assume that the four edges are $e_1 = (1, 2)$, $e_2 = (1, 2)$, $e_3 = (1, 3)$ and $e_4 = (2, 3)$. By Lemma 8, we can assume that algorithm $A$ has matched $e_1$ with $e_2$ in the first phase. We select $p_1 = \{e_1, e_2\}$ and $p_2 = \{e_3, e_4\}$ to be part of matching $\hat{M}$, with total cost $2 + 3 = 5$. Consider the case that $p_2$ was not matched by $A$ in the

<table>
<thead>
<tr>
<th>Type of quad</th>
<th>Number of edges in $E_F(O_i)$</th>
<th>Number of edges in $E_F(X)$</th>
<th>Number of edges in $E_F(C)$</th>
<th>Cost</th>
<th>Number of special edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td></td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1, 1</td>
<td>2</td>
<td></td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td></td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2 Overview of different types of quads with cost 4, containing at least 1 edge from $E_F(O_i)$. The entry “1,1” for quad type 3 means that there is one edge from $E_F(O_i)$ and one edge from $E_F(O_{i'})$ for $i \neq i'$.  

ISAAC 2018
first phase. (This is the more difficult case.) Assume that $A$ has matched $p_3 = \{e_3, x\}$ and $p_4 = \{e_4, y\}$, where $x$ and $y$ are edges not in $Q$. We add $(p_1, p_3)$ and $(p_1, p_4)$ to $H$. Each of these edges has savings at least 1, and thus they contribute 2 to $S_2$, or 1 to $\frac{1}{2}S_2$. We have $\phi_Q \leq 5 - 1 = 4 = \frac{4}{3} \text{cost}(Q)$. As $\phi_Q \leq \frac{4}{3} \text{cost}(Q)$ can be shown to hold also for all other cases of quads $Q$ in the optimal solution, algorithm $A$ is a $\frac{4}{3}$-approximation algorithm for $\text{PQ}(\#1 = 2)$.

For problem $\text{PQ}(\#1 = 2, \text{distinct})$, the $4k$ input vectors can be viewed as the edges of a simple graph. We follow the same approach as in the previous paragraph, but since a simple graph with four edges spans at least 4 nodes, we only need to consider cases where $\text{cost}(Q) \geq 4$. This allows us to show that $\phi_Q \leq \frac{13}{10} \text{cost}(Q)$ in all cases, implying that algorithm $A$ is a $\frac{13}{10}$-approximation algorithm for this problem.

4 Bad instances

In Section 4.1 we provide an instance that, together with the result in the previous section, yields the tight bound claimed for problem $\text{PQ}(\#1 = 2, \text{distinct}, \text{connected})$ in Theorem 13. We illustrate in Section 4.2 that a natural greedy algorithm (that can be seen as an alternative for algorithm $A$) has a worst-case ratio worse than the worst-case ratio of algorithm $A$. The instances that provide lower bound results for problem $\text{PQ}$ and the other special cases, as announced in Table 1, can be found in [6].

4.1 An instance of $\text{PQ}(\#1 = 2, \text{distinct}, \text{connected})$

Consider the instance $I$ consisting of the following 8 vectors $v_1, \ldots, v_8$.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Since each vector contains two 1’s, the vectors are pairwise distinct, and the induced graph is connected, this is an instance of $\text{PQ}(\#1 = 2)$, distinct, connected). The instance can be represented by the graph shown in Figure 2.
Figure 3 An instance of PQ($\#1 = 2$, distinct, connected).

The optimal solution for this instance has cost 8, with the two quads

\[
\{v_1, v_2, v_3, v_4\} = \{(1, 2), (1, 3), (2, 4), (3, 4)\},
\{v_5, v_6, v_7, v_8\} = \{(4, 5), (4, 6), (5, 7), (6, 7)\}.
\]

Algorithm $A$ may, in the first phase, construct a matching with cost 12 consisting of the following pairs:

\[
\{v_1, v_2\} = \{(1, 2), (1, 3)\}, \{v_3, v_5\} = \{(2, 4), (4, 5)\},
\{v_4, v_8\} = \{(3, 4), (4, 6)\}, \{v_7, v_8\} = \{(5, 7), (6, 7)\}.
\]

Any two pairs share at most 1 node. Hence, the total savings that can be made in the second matching are at most 2, so by Corollary 7 we have $A(I) \geq 10$. Hence, the worst-case ratio of $A$ is at least $10/8 = 5/4$.

Observation 16. For the instance depicted in Figure 2, $cost(A) = \frac{5}{4}OPT$.

Theorem 13 now follows from Lemma 14 and Observation 16.

4.2 Bad instances for a natural greedy algorithm

In this section, we show that the worst-case ratio of a natural greedy algorithm is worse than the worst-case ratio of algorithm $A$.

An informal description of the greedy algorithm for problem PQ (and its special cases) is as follows: repeatedly select, among all possible quads, a quad with lowest cost, and remove the vectors in the selected quad from the instance; stop when no more vectors remain.

Below we present an instance of problem PQ($\#1 = 2$, distinct, connected) showing that the worst-case performance of this greedy algorithm is worse than the worst-case performance of algorithm $A$. In [6] we present an instance of problem PQ with the same property.

An instance of PQ($\#1 = 2$, distinct, connected)

Consider the instance $I$ of PQ($\#1 = 2$, distinct, connected) consisting of 8 vectors represented in a graph shown in Figure 3 (recall that a vector in PQ($\#1 = 2$, distinct, connected) corresponds to an edge in a simple graph).

An optimal solution for this instance has cost 10, with the two quads \{(1, 2), (2, 5), (3, 5), (3, 4)\} and \{(6, 7), (5, 7), (5, 8), (8, 9)\}, each having cost 5. Since the instance features no quad with cost 4, the greedy algorithm may first select the following quad with cost 5: \{(2, 5), (3, 5), (5, 7), (5, 8)\}. Next, what remains is a quad of cost 8: \{(1, 2), (3, 4), (6, 7), (8, 9)\}.

Hence, the worst-case ratio of the greedy algorithm is at least $13/10$, which is larger than the $5/4$ approximation guarantee for algorithm $A$. 
5 Conclusion

We have studied the worst-case behavior of a natural algorithm for partitioning a given set of vectors into quadruples and shown the precise worst-case behavior of this algorithm for all cases except \(PQ(\#1 = 2, \text{distinct})\), where a small gap remains. It is a natural question to study an extension where we form clusters consisting of \(2^s\) vectors for some given integer \(s \geq 2\). Indeed, if we form groups of size \(2^s\) by running \(s\) rounds of matching, the worst-case ratio is easily seen to be bounded by \(2^s - 1\). To explain this, let \(M\) be the minimum-cost matching of the first round. Then \(A(I) \leq \text{cost}(M)\) and \(\text{OPT}(I) \geq \text{cost}(M)/2^s - 1\) as the cost of the optimum (viewed as being constructed in \(s\) rounds) is at least \(\text{cost}(M)\) after the first round and could then halve in each further round. Moreover, since we have shown that the cost of the algorithm after two rounds is at most \(\frac{3}{2}\) times the optimal cost after two rounds, we get a ratio of \(\frac{3}{2} \times 2^{s-2} = 3 \times 2^{s-3}\). We leave the question of finding the worst-case ratio for arbitrary \(s\) as an open problem.

References

Abstract

The fuzzy $K$-means problem is a popular generalization of the well-known $K$-means problem to soft clusterings. We present the first coresets for fuzzy $K$-means with size linear in the dimension, polynomial in the number of clusters, and poly-logarithmic in the number of points. We show that these coresets can be employed in the computation of a $(1 + \epsilon)$-approximation for fuzzy $K$-means, improving previously presented results. We further show that our coresets can be maintained in an insertion-only streaming setting, where data points arrive one-by-one.

2012 ACM Subject Classification Theory of computation → Unsupervised learning and clustering

Keywords and phrases clustering, fuzzy $k$-means, coresets, approximation algorithms, streaming

1 Introduction

Clustering is a widely used technique in unsupervised machine learning. The goal is to divide some set of objects into groups, the so-called clusters, such that objects in the same cluster are more similar to each other than to objects in other clusters. Nowadays, clustering is ubiquitous in many research areas, such as data mining, image and video analysis, information retrieval, and bioinformatics. The most common approach are hard clusterings, where the input is partitioned into a given number of clusters, i.e. each point belongs to exactly one of the clusters. The $K$-means problem is the most well-known hard clustering problem. It has been studied extensively from practical and theoretic points of view. However, in some applications it is beneficial to be less decisive and allow points to belong to more than one cluster. This idea leads to so-called soft clusterings. In the following, we study a popular soft clustering problem, the fuzzy $K$-means problem.

The fuzzy $K$-means objective function goes back to work by Dunn and Bezdek et al. [4, 10]. Today, it has found numerous practical applications, for example in data mining [19], image segmentation [27], and biological data analysis [9]. Practical applications generally use the fuzzy $K$-means algorithm, an iterative relocation scheme similar to Lloyd’s algorithm [25] for
$K$-means, to tackle the problem. The fuzzy $K$-means algorithm has been proven to converge to a local minimum or a saddle point of the objective function [4, 5]. Distinguishing whether the fuzzy $K$-means algorithm has reached a local minimum or a saddle point is a problem which got some attention on its own [20, 24]. Moreover, it is known that the algorithm converges locally, i.e. started sufficiently close to a minimizer, the iteration sequence converges to that particular minimizer [17]. However, from a theoretician’s point of view this algorithm has the major downside that stationary points of the objective function can be arbitrarily worse than a globally optimal solution [6]. Currently, the only paper on algorithms with approximation guarantees for the fuzzy $K$-means problem is [6], where the authors present a PTAS assuming a constant number of clusters.

Clustering is usually applied when huge amounts of data need to be processed. This has sparked significant interest in researching clustering in a streaming model, where the data does not fit into memory. A lot of research has been done on this setting for $K$-means. In a single pass setting, where we are only allowed to read the data set once, the $K$-means objective function can be approximated up to a constant factor, by choosing $O(K \log(K))$ means, instead of $K$ [1]. This has been improved to an algorithm computing exactly $K$ means but still maintaining a constant factor approximation [7, 28]. There, the authors considered a streaming setting where points arrive one-by-one and they are allowed to use $O(K \log(N))$ memory, where $N$ is the total number of points.

The goal of a coreset is to find a small representation of a large data set, retaining the characteristics of the original data. Coresets have emerged as a key technique to tackle the streaming model. The idea is to treat the computation of the coreset as an online problem where points arrive in some kind of stream. If, after having read the whole stream, the computed coreset is small enough to fit into memory, then standard algorithms can be used to solve the problem almost optimally for the points in the stream. Usually, the algorithm does not know the size of the stream beforehand and hence, always maintains a coreset of the points seen so far.

The first coreset construction for $K$-means is due to Har-Peled and Mazumdar, and is of size $O(\log(N))$ [16]. They also showed how to maintain a coreset, with size poly-logarithmic in $N$, of a data stream, by combining their notion of a coreset with the merge-and-reduce technique by Bentley and Saxe [3]. This construction was improved to a coreset with size independent of $N$ [15]. Feldman and Langberg presented a general framework computing coresets for a large class of hard clustering problems with size independent of $N$ [12]. Later, Feldman et al. presented coresets with size independent of $N$ and $D$ by using a construction based on low-rank approximation [14]. Furthermore, they generalize Har-Peled and Mazumdar’s application of the merge-and-reduce technique, showing how coresets with certain properties can be maintained in a streaming setting. The results of this paper are based on Chen’s sampling based construction, which yields coresets with size poly-logarithmic in $N$, $K$, and $D$ [8]. Applying the merge-and-reduce technique, Chen’s coresets can also be used to maintain a poly-logarithmic sized coreset of a data stream.

There has been some work on applying the fuzzy $K$-means algorithm to large data sets. Hore et al. [21] presented a single pass variant of the algorithm, which processes the data chunk-wise. This idea was refined and extended to a single pass and online kernel fuzzy $K$-means algorithm [18]. However, these are still variants of the fuzzy $K$-means algorithm, hence provide no guarantees for the quality of solutions. So far, no coreset constructions have been presented for the fuzzy $K$-means problem, and the literature is not rich on coreset constructions for soft clustering problems, in general. There is a construction for the problem of estimating mixtures of semi-spherical Gaussians which yields coresets with size independent of $N$ [11]. This result was generalized to a large class of hard and soft clustering problems based on $\mu$-similar Bregman divergences [26].
1.1 Our Result

We prove the existence of small coresets for the fuzzy K-means problem. In Section 3, we show that, by adjusting some parameters of Chen's construction [8], we obtain a coreset for the fuzzy K-means problem with size still poly-logarithmic in N. Our proof technique is a non-trivial combination of the notion of negligible fuzzy clusters [6] and weak coresets [13]. This results in a general weak-to-strong lemma (cf. Lemma 7), which states that weak coresets for the fuzzy K-means problem fulfilling certain conditions are already strong coresets. Afterwards, we argue that our adaptation of Chen’s algorithm yields a weak coreset satisfying all conditions of the weak-to-strong theorem (a comprehensive proof can be found in the full version). In Section 4, we substantiate the usefulness of our result by presenting two applications of coresets for fuzzy K-means. First, we improve the analysis of a previously presented [6] PTAS for fuzzy K-means, removing the dependency on the weights of the data points from the runtime. Running this algorithm on our coreset instead of the original input improves upon the runtime of previously known \((1 + \epsilon)\)-approximation schemes. The improvement lies in the exponential term, which we reduce from \(N^{O(\text{poly}(K,1/\epsilon))}\) to \(\log(N)^{O(\text{poly}(K,1/\epsilon))}\), while maintaining non-exponential dependence on \(D\). Second, we argue that an application of the merge-and-reduce technique enables us to maintain a fuzzy K-means coreset in a streaming model, where points arrive one-by-one.

2 Preliminaries

Let \(X \subseteq \mathbb{R}^D\) be a set of points in \(D\)-dimensional space and \(w : X \rightarrow \mathbb{N}\) be an integer weight function on the points. Using integer weights eases the notation of our exposition. We later argue how our results generalize to rational weights. Unweighted data sets are denoted by using the weight function \(1\) mapping every input to 1. We call \(w(X) = \sum_{x \in X} w(x)\) the total weight of \(X\) and denote the maximum and minimum weights by \(w_{\text{max}}(X) = \max_{x \in X} w(x)\) and \(w_{\text{min}}(X) = \min_{x \in X} w(x)\).

Definition 1 (Fuzzy K-means). Let \(m \in \mathbb{R}_{>1}\) and \(K \in \mathbb{N}\). The fuzzy K-means problem is to find a set of means \(M = \{\mu_k\}_{k \in [K]} \subseteq \mathbb{R}^D\) and a membership function \(r : X \times [K] \rightarrow [0, 1]\) minimizing
\[
\phi(X, w, M, r) = \sum_{x \in X} w(x) \sum_{k \in [K]} r(x, k)^m \| x - \mu_k \|^2
\]
subject to
\[
\forall x \in X : \sum_{k \in [K]} r(x, k) = 1 .
\]

The parameter \(m\) is called fuzzifier. It determines the softness of an optimal clustering and is not subject to optimization, since the cost of any solution can always be decreased by increasing \(m\). In the case \(m = 1\), the cost can not be decreased by assigning membership of a point to any mean except its closest. Consequently, optimal solutions of the fuzzy K-means problem for \(m = 1\) coincide with optimal solutions for the K-means problem on the same instance. Hence, in the following we always assume \(m\) to be some constant larger than 1.

Similar to the classic K-means problem, it is easy to optimize means or memberships of fuzzy K-means, assuming the other part of the solution is fixed [4]. This means, given some set of means \(M\) we call a respective optimal membership function \(r^\ast_M\) induced by \(M\) and set \(\phi(X, w, M) := \phi(X, w, M, r^\ast_M)\). Analogously, given some membership function \(r\) we call a respective optimal set of means \(M^\ast_r\) induced by \(r\) and set \(\phi(X, w, r) := \phi(X, w, M^\ast_r, r)\). Finally, given some optimal solution \(M^\ast, r^\ast\) we denote \(\phi_{\text{opt}}(X, w) := \phi(X, w, M^\ast, r^\ast)\).
2.1 Fuzzy Clusters

Recall that, in a soft-clustering, there is no partitioning of the input points. Instead, we describe the \(k\)th cluster of a fuzzy clustering as a vector of the fractions of points assigned to it by the membership function. We denote the size (or the total weight) of the \(k\)th cluster by \(r(X, w, k) = \sum_{x \in X} w(x)r(x,k)^m\). Given a set of means \(M\), we denote the cost of the \(k\)th cluster by \(\phi_k(X, w, M, r) = \sum_{x \in X} w(x)r(x,k)^m \|x - \mu_k\|^2\).

2.2 \(K\)-Means Notation

We denote the distance of a point to a set of means \(M\) by \(d(x, M) = \min_{\mu \in M}\{\|x - \mu\|\}\) and the \(K\)-means cost by \(km(X, w, M) = \sum_{x \in X} w(x)d(x, M)^2\). Let \(C \subseteq X\) be some cluster, then \(km(C, w) = \sum_{x \in C} w(x)\|x - \mu_w(C)\|^2\), where \(\mu_w(C) = \sum_{x \in C} w(x)x/w(C)\).

3 Coresets for Fuzzy \(K\)-Means

A coreset is a representation of a data set that preserves properties of the original data set [16]. Formally, we require the cost of a set of means with respect to the coreset to be close to the cost the same set of means incurs on the original data.

Definition 2 (Coreset). Let \(\epsilon \in (0, 1)\). A set \(S \subseteq \mathbb{R}^D\) together with a weight function \(w_S : S \rightarrow \mathbb{N}\) is called an \(\epsilon\)-coreset of \((X, w)\) for the fuzzy \(K\)-means problem if

\[
\forall M \subseteq \mathbb{R}^D, |M| \leq K : \phi(S, w_S, M) \in [1 \pm \epsilon]\phi(X, w, M) ,
\]

We sometimes refer to a coreset as a strong coreset.

In the following, we show how to construct coresets for the fuzzy \(K\)-means problem with high probability. To this end, our proof consists of two independent steps. First, we show that it is sufficient to construct a so-called weak coreset [13] for the fuzzy \(K\)-means problem fulfilling certain properties. Second, we present an adaptation of Chen’s coreset construction for \(K\)-means [8] which computes weak coresets with the desired properties, with high probability.

Theorem 3. There is an algorithm that, given a set \(X \subseteq \mathbb{R}^D, K \in \mathbb{N}, \delta \in (0, 1)\), and \(\epsilon \in (0, 1)\), computes an \(\epsilon\)-coreset \((S, w_S)\), with \(S \subseteq X\) and \(w_S : S \rightarrow \mathbb{N}\), of \((X, 1)\) for the fuzzy \(K\)-means problem, with probability at least \(1 - \delta\), such that

\[
|S| \in \mathcal{O}\left(\log(N)\log(\log(N))^2\epsilon^{-3}DK^{km-1}\log(\delta^{-1})\right) .
\]

The algorithms’ runtime is \(\mathcal{O}(NDK\log(\delta^{-1}) + |S|)\).

This result trivially generalizes to integer weighted data sets, by treating each point \(x \in X\) as \(w(x)\) copies of the same point. However, in that case we have to replace each occurrence on \(N\) in the runtime of the algorithm and the size of the coreset by \(w(X)\). For rational weights, we normalize the weight function. This incurs an additional multiplicative factor of \(w_{\text{max}}(X)/w_{\text{min}}(X)\) to each occurrence of \(N\).

3.1 From Weak to Strong Coresets

Weak coresets are a relaxation of the previously introduced (strong) coresets. Consider a set of points together with a weight function and a set of solutions. This forms a weak coreset if the set of solutions contains a solution close to the optimum and the coreset property (1) is satisfied for all solutions from the solution set.
Definition 4 (Weak Coresets). A set \( S \subset \mathbb{R}^D \) together with a weight function \( w_S : S \to \mathbb{N} \) and a set of solutions \( \Theta \subseteq \{ \theta : \theta \subset \mathbb{R}^D, |\theta| \leq K \} \) is called a weak \( \epsilon \)-coreset of \((X, w)\) for the fuzzy \( K \)-means problem if

\[
\exists M \in \Theta : \phi(S, w_S, M) \leq (1 + \epsilon) \cdot \phi^{opt}(X, w) \quad \text{and} \quad \forall M \in \Theta : \phi(S, w_S, M) \in [1 \pm \epsilon] \phi(X, w, M).
\]

In contrast to the definition of weak coresets for the \( K \)-means problem [13], we consider elements \( M \) of a given set of solutions \( \Theta \) instead of subsets of a set of candidate means. This is just a slight generalization which allows us to characterize solutions more precisely.

One difficulty when analysing the fuzzy \( K \)-means objective function is that, in optimal solutions, clusters are never empty. Consider a set of means, where there exists a mean which is far away from every point. In an optimal hard clustering, this mean’s cluster is empty and we can safely ignore it in the analysis. For fuzzy \( K \)-means, this is not the case. In an optimal solution, every point has a non-trivial membership to this mean, thus it cannot be ignored (or removed from the solution) without increasing the cost. Bounding the cost of means with small membership mass proves to be rather difficult. A central concept we use to control the cost of such means are fuzzy clusters which are almost empty, or negligible.

Definition 5 (negligible). Let \( M \subset \mathbb{R}^D \) with \( |M| \leq K \). We say the \( k \)th cluster of a membership function \( r : X \times [|M|] \to [0, 1] \) is \((K, \epsilon)\)-negligible if

\[
\forall x \in X : r(x, k) \leq \frac{\epsilon}{4nK^2}.
\]

In the following, we omit the parameters \((K, \epsilon)\) if they are clear from context.

We cannot preclude the possibility that an optimal fuzzy \( K \)-means clustering contains a negligible cluster. However, we can circumvent negligible clusters altogether, by observing that we can remove a mean inducing a negligible cluster without increasing the cost significantly.

Theorem 6 ([6]). Let \( M \subset \mathbb{R}^D \) with \( |M| \leq K \) and \( \epsilon \in (0, 1) \). There exists a set of means \( M' \subset M \) with

\[
\phi(X, w, M') \leq (1 + \epsilon)\phi(X, w, M),
\]

such that the optimal membership function with respect to \( M' \) contain no negligible clusters.

Given some set of means, the optimal memberships of a point depend only on the location of the point relative to the means and not on its weight or any other points in the data set [4]. This means that negligible clusters are, in some sense, transitive. That is: If a cluster induced by some set of means is negligible, then it is also negligible with respect to any subset of \( X \) and the same set of means. Using this observation we can prove our key weak-to-strong result.

Lemma 7 (weak-to-strong). Let \( \epsilon \in (0, 1) \) and

\[
\Theta_{(K, \epsilon)}(X) := \left\{ M \subset \mathbb{R}^D \mid |M| \leq K \text{ and } M \text{ induces no negligible cluster with respect to } X \right\}.
\]

If \( S \subset X \) and \( w_S : S \to \mathbb{N} \), such that \((S, w_S, \Theta_{(K, \epsilon)}(X))\) is weak \( \epsilon \)-coreset of \((X, w)\) for the fuzzy \( K \)-means problem, then \((S, w_S)\) is a strong \((3\epsilon)\)-coreset of \((X, w)\) for the fuzzy \( K \)-means problem.
Proof. We need to verify that the coreset property (1) holds for all solutions \( M \subset \mathbb{R}^D \) with \(|M| \leq K\). Since \((S, w_S, \Theta_{(K,\epsilon)}(X))\) is a weak \(\epsilon\)-coreset we only have to show this for all \(M \notin \Theta_{(K,\epsilon)}(X)\). From Theorem 6, we know that there exists \(M' \in \Theta_{(K,\epsilon)}(X)\), \(M' \subseteq M\) with \(\phi(X, w, M') \leq (1 + \epsilon)\phi(X, w, M)\).

We obtain the upper bound by observing that

\[
\phi(S, w_S, M) \leq \phi(S, w_S, M') \quad (M' \subseteq M)
\]

\[
\leq (1 + \epsilon)\phi(X, w, M') \quad \text{(weak coreset property)}
\]

\[
\leq (1 + \epsilon)^2\phi(X, w, M) \quad \text{(choice of } M')
\]

\[
\leq (1 + 3\epsilon)\phi(X, w, M) \quad (\epsilon \in (0, 1))
\]

The lower bound is slightly more involved. Again, from Theorem 6, we obtain that there exists \(M_S^0 \in \Theta_{(K,\epsilon)}(S)\), \(M_S^0 \subseteq M\) with \(\phi(S, w_S, M_S^0) \leq (1 + \epsilon)\phi(S, w_S, M)\). Recall that, for each point, the membership induced by some set of means only depends on the point itself and the given set of means. In particular, this membership does not depend on the weight of the point, nor on other data points. Hence, if there is no point in \(X\) such that the induced membership with respect to some mean \(\mu_k \in M\) is larger than some constant, then there is no point in \(S \subseteq X\), such that the induced membership to \(\mu_k \in M\) is larger than this constant. Since \(M' \in \Theta_{(K,\epsilon)}(X)\), it holds that all means in \(M \setminus M'\) induce negligible clusters on \(S\) and thus \(M_S^0 \subseteq M'\). We conclude

\[
\phi(S, w_S, M) \geq \frac{1}{1 + \epsilon} \phi(S, w_S, M_S^0) \quad \text{(choice of } M_S^0)
\]

\[
\geq \frac{1}{1 + \epsilon} \phi(S, w_S, M') \quad (M_S^0 \subseteq M')
\]

\[
\geq \frac{1 - \epsilon}{1 + \epsilon} \phi(X, w, M') \quad \text{(weak coreset property)}
\]

\[
\geq \frac{1 - \epsilon}{1 + \epsilon} \phi(X, w, M) \quad (M' \subseteq M)
\]

\[
\geq (1 - 3\epsilon)\phi(X, w, M) \quad (\epsilon \geq 0)
\]

\(\blacksquare\)

3.2 Weak Coresets for Solutions with Non-Negligible Clusters

In the following, we explain how to adapt Chen’s coreset construction for the \(K\)-means problem [8] to construct a set \(S \subseteq X\) and weight function \(w_S : S \rightarrow \mathbb{N}\) such that \((S, w_S, \Theta_{(K,\epsilon)}(X))\) is a weak \(\epsilon\)-coreset of \((X, 1)\) for the fuzzy \(K\)-means problem. Applying Lemma 7 to this construction yields Theorem 3. We give a high-level description of Chen’s algorithm. In the first step, we compute an \((\alpha, \beta)\)-bicriteria approximation of the \(K\)-means problem with respect to \(X\), i.e. a set \(M\) approximating an optimal \(K\)-means solution within factor \(\alpha\) and with \(|M| \leq \beta K\), such that \(\alpha, \beta \in \mathcal{O}(1)\).

In the second step, the input points are partitioned based on concentric balls around the means of the bicriteria approximation with exponentially increasing radii. By \(X_{i,j}\), we denote the intersection of \(X\) with the \(j^{th}\) annulus around the \(i^{th}\) mean. Then, we sample points from each \(X_{i,j}\) uniformly and independently at random. Finally, each point sampled from \(X_{i,j}\) is evenly weighted, such that the sum of these weights is equal to the number of original data points in \(X_{i,j}\). These sampled points together with the weights form the coreset.
There is no natural adaptation of the first step to fuzzy $K$-means since, so far, there exists no bicriteria approximation algorithm for the fuzzy $K$-means problem with constant $\alpha$ and $\beta$. However, we know that the $K$-means cost of all sets of means $M$ is no larger than $|M|^{m-1}$ times the fuzzy $K$-means cost of $M$ [6]. Hence, an $(\alpha, \beta)$-bicriteria approximation for the $K$-means problem is an $(\alpha \cdot (\beta K)^{m-1}, \beta)$-bicriteria approximation for the fuzzy $K$-means problem on the same instance. We can counteract this very coarse bound on the cost in the second step by sampling roughly a factor of $K^{O(m)}$ more points than the original algorithm.

Lemma 8. The algorithm described in the previous paragraph computes $S \subseteq X$ and $w_S : S \to \mathbb{N}$ such that $(S, w_S, \Theta_{(K,\epsilon)}(X))$ is a weak $\epsilon$-coreset of $(X, 1)$ for the fuzzy $K$-means problem, with high probability.

Proof Sketch. Let $M \in \Theta_{(K,\epsilon)}(X)$ be a set of means inducing no negligible clusters. We consider large balls around each mean of the bicriteria-approximation. As in Chen’s original proof, we establish the coreset property for the case where at least one mean of a given solution is outside of these balls and the case where all means are contained in the union of these balls.

For the first case, assume that $M$ contains at least one mean, say $\mu_k$, outside of (sufficiently large) balls around the means of the bicriteria approximation. Since $\mu_k$ has a non negligible portion of the membership of at least one point from which it is far away, we can bound the cost of $M$ from below. This lower bound is significantly larger than the distances of data points to their respective representative in the coreset. Using this, we can easily verify the coreset property with respect to $M$.

For the second case, assume that all means of $M$ lie in the union of these balls. In this case, we do not need to use that clusters induce non-negligible memberships. Instead, we can basically follow the arguments of Chen’s original proof. However, the cost estimations are more technically involved due to the difficult structure of the fuzzy $K$-means objective function. A detailed exposition of our proof can be found in the full version.

The size of the coreset and the runtime of the algorithm are as claimed in Theorem 3.

4 Applications

In the following, we present two applications of our coresets for fuzzy $K$-means. In general, our coresets can be plugged in before any application of an algorithm that tries to solve fuzzy $K$-means and can handle weighted data sets. If the applied algorithm’s runtime does not depend on the actual weights, then this leads to a significant reduction in runtime. We show that this yields a faster PTAS for fuzzy $K$-means than the ones presented before [6]. Furthermore, we argue that our coresets can be maintained in an insertion-only streaming setting.

4.1 Speeding up Approximation

We start by presenting an improved analysis of a simple sampling-based PTAS for the fuzzy $K$-means problem. Our analysis exploits that the algorithm can ignore the weights of the data points and still obtain an approximation guarantee of $(1 + \epsilon)$ for the weighted problem. This means, that the algorithm’s runtime is independent of the weights, and thus can be significantly reduced by applying it to a coreset instead of the original data. The first ingredient is the following, previously presented, soft-to-hard lemma.
**Algorithm 1:** Derandomized Sampling.

**Input:** $X \subset \mathbb{R}^D$, $K \subset \mathbb{N}$, $\varepsilon \in (0, 1)$

1. $\mathcal{T} \leftarrow \{ \mu_y(S) \mid S \subseteq X, |S| = \frac{64K}{\varepsilon} \}$ 
   
   /* $S$ as multisets - Points can occur multiple times in each $S$ and are counted with multiplicity. */

2. $M \leftarrow \arg\min_{T \subseteq \mathcal{T}, |T| = K} \{ \phi(X, w, T) \}$

3. Return $M$.

**Lemma 9** ([6]). Let $\varepsilon \in (0, 1)$, $r : X \times [K] \to [0, 1]$ be a membership function and let $M^*_r$ be a set of means induced by $r$.

If $\forall k \in [K] : r(X, w, k) \geq 16Kw_{\max}(X)/\varepsilon$, then there exist pairwise disjoint sets $C_1, \ldots, C_K \subseteq X$ such that for all $k \in [K]$:

$$w(C_k) \geq \frac{r(X, w, k)}{2},$$

$$\|\mu_w(C_k) - \mu_k\|^2 \leq \frac{\varepsilon}{r(X, w, k)} \phi_k(X, w, M^*_r, r),$$

and

$$km(C_k) \leq 4K \cdot \phi_k(X, w, M^*_r, r).$$

We combine this with a classic concentration bound by Inaba et al.

**Lemma 10** ([22]). Let $P \subset \mathbb{R}^D$, $n \in \mathbb{N}$, $\delta \in (0, 1)$, and let $S$ be a set of $n$ points drawn uniformly at random from $P$. Then we have

$$\Pr \left( \left\| \phi_1(S) - \phi_1(P) \right\|^2 \leq \frac{1}{\delta n} \frac{km(P, 1)}{|P|} \right) \geq 1 - \delta.$$ 

**Corollary 11.** Let $X \subset \mathbb{R}^D$, $w : X \to \mathbb{N}$, $K \subset \mathbb{N}$, $\varepsilon \in (0, 1)$, and let $C_1, \ldots, C_K \subseteq X$ be non-empty subsets of $X$. There exist $K$ multisets $S_1, \ldots, S_K \subseteq X$, such that

$$\forall k \in [K] : |S_k| = \frac{2}{\varepsilon} \text{ and } \|\mu_1(S_k) - \mu_w(C_k)\|^2 \leq \frac{\varepsilon km(C_k, w)}{w(C_k)}.$$ 

We can find means of subsets obtained from applying the soft-to-hard lemma to the clusters of an optimal fuzzy $K$-means solution by derandomizing Inaba’s sampling technique.

**Theorem 12.** Algorithm 1 computes $M \subset \mathbb{R}^D$ with $|M| = K$, such that

$$\phi(X, w, M) \leq (1 + \varepsilon)\phi^{opt}(X, w)$$

in time $DN^{O(K^2/\varepsilon)}$.

**Proof.** We analyse the result $M$ of Algorithm 1. Let $M^*, r^*$ be an optimal solution to the fuzzy $K$-means problem on $X, w$. Let $X_c$ be a modified point set, which contains $c$ copies of every point $x \in X$, where

$$c = \left\lceil \frac{\gamma \epsilon K w_{\max}(X)}{\epsilon \min_{k \in [K]} r^*(X, w, k)} \right\rceil,$$

for some large enough constant $\gamma$. For all sets of means $M$ and all membership functions $r$, we have $\phi(X_c, w, M, r) = c \cdot \phi(X, w, M, r)$. Thus, $M^*$ and $r^*$ (where $r^*(y, k) = r^*(x, k)$ for
all \( k \in [K] \) and \( x \in X, y \in X_c \) with \( x = y \) are also optimal for the modified instance \( X_c \). Observe, that for all \( k \in [K] \) we have
\[
    r^*(X_c, w, k) \geq \sum_{x \in X} \frac{\gamma K w_{max}(X) \min_{k \in [K]} r^*(X, w, k) w(x) r^*(x, k)^m}{\epsilon} \geq \frac{\gamma K w_{max}(X)}{\epsilon} \geq \frac{64 K w_{max}(X)}{\epsilon} .
\]
Observe, that \( M^* \) is a set of means induced by \( r^* \). Hence, by applying Lemma 9 with respect to \( X_c, w, r^* \), and \( \epsilon/4 \) we obtain that there exist disjoint sets \( C_1, \ldots, C_K \subseteq X_c \) such that for all \( k \in [K] \) we have
\[
    w(C_k) \geq \frac{r^*(X_c, w, k)}{2}, \quad \|\mu_w(C_k) - \mu_k\|^2 \leq \frac{\epsilon}{4r^*(X_c, w, k)} \phi_k(X_c, w, M^*, r^*), \quad \text{and}
\]
\[
    km(C_k, w) \leq 4 K \cdot \phi_k(X_c, w, M^*, r^*).
\]
Next, we apply Corollary 11 to \( X_c, w, K, \epsilon/(32K) \), and \( C_1, \ldots, C_K \). We obtain that there exist \( S_1, \ldots, S_K \subseteq X_c \) such that for all \( k \in [K] \) we have \( |S_k| = 64 K / \epsilon \) and
\[
    \|\mu_1(S_k) - \mu_w(C_k)\|^2 \leq \epsilon/(32K) km(C_k, w) / w(C_k).
\]
Since \( X_c \) consists of copies of points from \( X \), we conclude that \( S_1, \ldots, S_K \subseteq X \), if we treat the \( S_k \) as multisets, i.e. allow the same point to appear multiple times in the same set. Hence, by choice of \( M \), as made by Algorithm 1, we have \( \phi(X, w, M) \leq \phi(X, w, \{\mu_1(S_k)\}_{k \in [K]}) \).

Plugging all this together, we can bound the cost of \( M \) as follows
\[
\phi(X, w, M) \leq \phi(X, w, \{\mu_1(S_k)\}_{k \in [K]}) = \frac{1}{c} \phi(X_c, w, \{\mu_1(S_k)\}_{k \in [K]})
\]
\[
\leq \frac{1}{c} \phi(X_c, w, \{\mu_1(S_k)\}_{k \in [K]}, r^*) = \frac{1}{c} \sum_{x \in X_c} \sum_{k \in [K]} w(x) r^*(x, k)^m \|x - \mu_1(S_k)\|^2
\]
\[
\leq \phi(X, w, r^*) + \frac{2}{c} \sum_{x \in X_c} \sum_{k \in [K]} w(x) r^*(x, k)^m \|\mu_k - \mu_w(C_k)\|^2
\]
\[
+ \frac{2}{c} \sum_{x \in X_c} \sum_{k \in [K]} w(x) r^*(x, k)^m \|\mu_w(C_k) - \mu_1(S_k)\|^2
\]
(by 2-approximate triangle inequality)
\[
\leq \phi^{opt}(X, w) + \frac{\epsilon}{2c} \sum_{k \in [K]} \phi_k(X_c, w, M^*, r^*) \quad \text{(by (3))}
\]
\[
+ \frac{\epsilon}{c16K} \sum_{k \in [K]} \frac{km(C_k, w)}{w(C_k)} \sum_{x \in X_c} w(x) r^*(x, k)^m \quad \text{(by (5))}
\]
\[
\leq (1 + \epsilon/2) \phi^{opt}(X, w) + \frac{\epsilon}{2c} \sum_{k \in [K]} \phi_k(X_c, w, M^*, r^*) \quad \text{by (2) and (4))}
\]
\[
= (1 + \epsilon) \phi^{opt}(X, w).
\]

Bounding the runtime of Algorithm 1 is straightforward. We have to evaluate the cost of \( |T|^K \) different fuzzy K-means solution, each evaluation costing \( O(NDK) \). Hence, the total runtime is bounded by \( O(NDK |T|^K) = O(NDK(N^{64K/\epsilon})^K) = DN^{O(K^2/\epsilon)}. \)

Recall, that the runtime of Algorithm 1 is independent of point weights. Hence, we obtain a more efficient algorithm by first computing a coreset using Theorem 3 and then applying Algorithm 1 to this coreset instead of the original data set. In the following, we formally only state an unweighted version of our result.
Theorem 13. There exists an algorithm which, given \( X \subset \mathbb{R}^D \), \( K \in \mathbb{N} \), and \( \epsilon \in (0,1) \), computes a set \( M \subset \mathbb{R}^D \) with \( |M| = K \), such that with constant probability

\[
\phi(X, 1, M) \leq (1 + \epsilon)\phi^{opt}(X, 1)
\]

in time \( O(NDK) + (\log(N)D)^{O(K^2/(\epsilon \log(K/\epsilon)))} \).

Proof. Given \( X, K \), and \( \epsilon \), apply Theorem 3 (with \( \epsilon/3 \)) to obtain, with constant probability, an \( \epsilon/3 \)-coreset \( (S, w_S) \) of \( (X, 1) \). Let \( M \) be the output of Algorithm 1 given \( S, w_S \), and \( \epsilon/3 \) and let \( M_X^* \) be an optimal set of means with respect to \( X \). We obtain

\[
\phi(S, w_S, M) \leq (1 + \epsilon/3)\phi^{opt}(S, w_S) \leq (1 + \epsilon/3)\phi(S, w_S, M_X^*) \leq (1 + \epsilon/3)^2\phi^{opt}(X, 1) \leq (1 + \epsilon)\phi^{opt}(X, 1).
\]

The overall runtime is \( O(NDK) + D(|S|)^{O(K^2/(\epsilon \log(K/\epsilon)))} = O(NDK) + (\log(N)D)^{O(K^2/(\epsilon \log(K/\epsilon)))} \). ▷

The algorithm from Corollary 13 can also be applied to weighted data sets. However, its runtime is not independent of these weights. We argued that the runtime of the PTAS from Theorem 12 is independent of any weights, but this is not true for the coreset construction. Hence, weight functions have an impact on the runtime as discussed in Section 3 in regard to the coreset construction.

Nonetheless, our algorithm has significant advantages over previously presented \((1 + \epsilon)\)-approximation algorithms for fuzzy \( K \)-means. The runtimes of all algorithms presented in [6] have an exponential dependency on the dimension \( D \) or contain a term \( N^{O(poly(K,1/\epsilon))} \). Our result constitutes the first algorithm with a non-exponential dependence on \( D \) whose only exponential term is of the form \( \log(N)^{O(poly(K,1/\epsilon))} \).

Strictly speaking, applying Algorithm 1 directly to \( X \) is faster if \( D \in \Omega(N) \). However, in that case we can apply the lemma of Johnson and Lindenstrauss [23] to replace \( D \) by \( \log(N)/\epsilon^2 \).

4.2 Streaming Model

We give a brief overview of the method to maintain coresets in a streaming model presented in [14]. It is an improved version of the techniques previously used by [8] and [16]. The central observation is that the union of coresets of two input data sets is a coreset of the union of the data sets. Whenever a sufficient (depending on the coreset construction) number of points has arrived in the stream, we compute a coreset of these points. After two coresets have been computed, we merge them into a larger coreset of all points that have arrived, so far. Following two of these merge operations, we merge the two larger coresets into one even larger one. This continues in the fashion of a binary tree. Since our coresets for fuzzy \( K \)-means fulfil all requirements to apply this approach, it can also be used to maintain fuzzy \( K \)-means coresets in the streaming model.

Theorem 14. Given \( N \) data points in a stream (one-by-one) and \( \epsilon \in (0,1) \) one can maintain, with high probability, an \( \epsilon \)-coreset for the fuzzy \( K \)-means problem, of the points seen so far, using \( O(DK^{4m-1} \cdot \text{polylog}(N/\epsilon)) \) memory. Arriving data points cause an update with an amortized runtime of \( O(DK \cdot \text{polylog}(NDK/\epsilon)) \).
We proved that a parameter tuned version of Chen’s construction yields the first coresets for the fuzzy $K$-means problem. While there are a plethora of coreset constructions for $K$-means, Chen’s construction is the best purely sampling based approach. More efficient techniques, for example $\epsilon$-nets [15] or subspace approaches like low-rank approximation [14], heavily rely on the partitioning of the input set that a $K$-means solution induces. So far, we have not found a way to apply these to the, already notoriously hard to analyse, fuzzy $K$-means objective function. This is because the membership function essentially introduces an unknown weighting on the points. Hence, when the data set is partitioned or projected into some subspace without respecting this weighting, we introduce a factor $K\mathcal{O}(1)$ to the cost estimation. It has proven difficult to control these additional factors. Partly for these reasons, there is still a large number of open questions regarding fuzzy $K$-means.

In this paper, we almost match the asymptotic runtime of the fastest $(1+\epsilon)$-approximation algorithms for $K$-means. However, even assuming constant $K$, our algorithms lack practicality due to the large constants hidden in the $\mathcal{O}$. Hence, this raises interesting follow-up questions. Is there an efficient approximation algorithm for fuzzy $K$-means with a constant approximation factor? What can be done in terms of bicriteria algorithms, i.e. if we are allowed to chose more than $K$ means? In regard to the complexity of fuzzy $K$-means it is interesting to examine whether one can show that there is no true PTAS (polynomial runtime in $N$, $D$, and $K$) for fuzzy $K$-means, as it was shown for $K$-means [2]. Finally, can we relate the hardness of fuzzy $K$-means directly to $K$-means?

References


46:12 Coresets for Fuzzy $K$-Means with Applications


Streaming Algorithms for Planar Convex Hulls

Martín Farach-Colton
Department of Computer Science, Rutgers University, Piscataway, USA
farach@cs.rutgers.edu

Meng Li
Department of Computer Science, Rutgers University, Piscataway, USA
ml910@cs.rutgers.edu

Meng-Tsung Tsai
Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan
mtsai@cs.nctu.edu.tw

Abstract

Many classical algorithms are known for computing the convex hull of a set of \( n \) point in \( \mathbb{R}^2 \) using \( O(n) \) space. For large point sets, whose size exceeds the size of the working space, these algorithms cannot be directly used. The current best streaming algorithm for computing the convex hull is computationally expensive, because it needs to solve a set of linear programs.

In this paper, we propose simpler and faster streaming and W-stream algorithms for computing the convex hull. Our streaming algorithm has small pass complexity, which is roughly a square root of the current best bound, and it is simpler in the sense that our algorithm mainly relies on computing the convex hulls of smaller point sets. Our W-stream algorithms, one of which is deterministic and the other of which is randomized, have nearly-optimal tradeoff between the pass complexity and space usage, as we established by a new unconditional lower bound.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases Convex Hulls, Streaming Algorithms, Lower Bounds

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.47


1 Introduction

The convex hull of a set \( P \) of points in \( \mathbb{R}^2 \) is the smallest convex set that contains \( P \). We denote the convex hull of \( P \) by \( \text{conv}(P) \) and denote the set of extreme points in \( \text{conv}(P) \) by \( \text{ext}(P) \). Let \( n = |P| \) and \( h = |\text{ext}(P)| \). Note that \( h \leq n \) because \( \text{ext}(P) \) is a subset of \( P \). By computing the convex hull of \( P \), we mean outputting the points in \( \text{ext}(P) \) in clockwise order.

There is a long line of research on computing the convex hull using \( O(n) \) space. In the RAM model, Graham [20] gave the first algorithm, called the Graham Scan, with running time \( O(n \log n) \). Subsequently, several algorithms were devised with the same running time, but with different approaches [2, 6, 26, 34]. In the output-sensitive model, where the running time depends on \( n \) and \( h \), Jarvis [25] proposed the Gift Wrapping algorithm, which has

---

1 This research was supported in part by NSF CCF 1637458, NIH 1 U01 CA198952-01, a NetAPP Faculty Fellowship and a gift from Dell/EMC.

2 This research was supported in part by the Ministry of Science and Technology of Taiwan under contract MOST grant 107-2218-E-009-026-MY3, and the Higher Education Sprout Project of National Chiao Tung University and Ministry of Education (MOE), Taiwan.
running time \(O(nh)\). This algorithm was later improved by Kirkpatrick and Seidel [28] and Chan [12], both of which achieve running time of \(O(n \log h)\). In the online model, where input points are given one by one and algorithms need to compute the convex hull of points seen so far, Overmars and van Leeuwen’s algorithm [33] can update the convex hull in \(O(log^2 n)\) time per incoming point. Brodal and Jacob [9] reduced the update time to \(O(\log n)\).

**Streaming Model.** The algorithms mentioned above all require \(s = \Omega(n)\) working space (memory) in the worst case. Therefore, none of these can handle the case when \(s \ll n\), that is, when either \(n\) is very large (a massive data set) or \(s\) is very small (such as in embedded systems). In order to explore the convex hull problem with such a memory restriction, we consider the standard streaming models [5,15,16,32,36], where the given points are stored on a read-only or writable tape in an arbitrary order. If the tape is read-only, then the model is simply called the streaming model [5,32]. Otherwise the tape is writable, and the model is called the W-stream model [15,16,36]. We refer to algorithms in the streaming model as streaming algorithms and algorithms in the W-stream model as W-stream algorithms. In both models, algorithms can manipulate the working space while reading the points sequentially from the beginning of the tape to the end; however, only algorithms in the W-stream model can modify the tape, detailed in Section 4. Hence, algorithms in this model cannot access the input randomly, which is different from the model for in-place algorithms [8,10]. The extreme points are written to a write-only stream. The pass complexity of an algorithm refers to the number of times the algorithms scans the tape from the beginning to the end. The goal is to devise streaming and W-stream algorithms that have small pass and space complexities.

No single-pass streaming algorithm can compute the convex hull using \(o(n)\) space because it is no easier than sorting \(n\) positive numbers in \(\mathbb{R}\). Since sorting \(n\) numbers using \(s\) spaces requires \(\Omega(n/s)\) passes [31], computing the convex hull in a single pass requires linear space. However, Chan and Chen [13] showed that the space requirement can be significantly reduced if multi-pass algorithms are allowed. Specifically, their streaming algorithm uses \(O(\delta^{-2})\) passes, \(O(\delta^{-2}hn^3)\) space, and \(O(\delta^{-2}n \log n)\) time for any constant \(\delta \in (0,1)\). On the other hand, to have small space complexity, one can appeal to a general scheme to convert PRAM algorithms to W-stream algorithms established by Demetrescu et al. [15], summarized in Section 4. Using this technique yields a W-stream algorithm that uses \(O((n/s) \log h)\) passes and \(O(s)\) space where \(s\) can be as small as constant.

**Our Contribution.** We devise a new \(O(n \log h)\)-time RAM algorithm to compute the convex hull (Section 2). Then, we adapt the RAM algorithm to both models.

In the streaming model, the pass complexity of our algorithm is roughly a square root of that of Chan and Chen’s algorithm [13] if both have the same space usage. We have:

> **Theorem 1.** Given a set \(P\) of \(n\) points in \(\mathbb{R}^2\) on a read-only tape where \(|\text{ext}(P)| = h\), there exists a deterministic streaming algorithm to compute the convex hull of \(P\) in \(O(\delta^{-1})\) passes using \(O(\min\{\delta^{-1}hn^3 \log n, n\})\) space and \(O(\delta^{-2}n \log n)\) time for every constant \(\delta \in (0,1)\).

In the W-stream model, we adapt the RAM algorithm to two W-stream algorithms. One uses \(O(s)\) space for any \(s = \Omega(\log n)\) and the other uses \(O(s)\) space for any \(s = \Omega(1)\). The pass complexity of our W-stream algorithms are \(O([h/s] \log n)\) and \(O(h/s + \log n)\), which are smaller than \(O((n/s) \log h)\), the best pass complexity among those W-stream algorithms that are converted from PRAM algorithms in algebraic decision tree model [15], when \(s \leq h\).
The first W-stream algorithm is deterministic, and we get:

**Theorem 2.** Given a set $P$ of $n$ points in $\mathbb{R}^2$ where $|\text{ext}(P)| = h$, there exists a deterministic W-stream algorithm to compute the convex hull of $P$ in $O(\lceil h/s \rceil \log n)$ passes using $O(s)$ space and $O(n \log^2 n)$ time for any $s = \Omega(\log n)$.

Next, we randomize the above W-stream algorithm. A logarithmic factor can be shaved off from the pass complexity with probability $1 - 1/n^{\Omega(1)}$, abbreviated as w.h.p. We have:

**Theorem 3.** Given a set $P$ of $n$ points in $\mathbb{R}^2$ where $|\text{ext}(P)| = h$, there exists a randomized W-stream algorithm to compute the convex hull of $P$ in $p$ passes using $O(s)$ space and $O(n \log^2 n)$ time for any $s = \Omega(1)$, where $p = O(h/s + \log n)$ w.h.p.

We prove that our W-stream algorithms have nearly-optimal tradeoff between pass and space complexities by showing Theorem 4, which generalizes Guha and McGregor’s lower bound (Theorem 8 in [22]). We remark that this lower bound is sharp because it matches the bounds of our randomized W-stream algorithm when $h = \Omega(s \log n)$.

**Theorem 4.** Given a set $P$ of $n$ points in $\mathbb{R}^2$ where $|\text{ext}(P)| = h = \Omega(1)$, any streaming (or W-stream) algorithm that computes the convex hull of $P$ with success rate $\geq 2/3$, and uses $s$ bits requires $\Omega(\lceil h/s \rceil)$ passes.

We note here that space is measured in terms of bits for lower bounds and in terms of points for upper bounds. This asymmetry is a common issue for geometric problems because most geometric problems are analyzed under the RealRAM model, where precision of points (or other geometric objects) is unbounded.

**Applications.** Our W-stream algorithms can handle the case for $s \leq h$ because it outputs extreme points on the fly. This output stream can be used as an input stream for another streaming algorithm, such as for diameter [37] and minimum enclosing rectangle [38], both of which rely on Shamos’ rotating caliper method [37]. Theorems 2 and 3 imply Corollary 5.

**Corollary 5.** Given a set $P$ of $n$ points in $\mathbb{R}^2$ where $|\text{ext}(P)| = h$, there exists a deterministic W-stream algorithm to compute the diameter and minimum enclosing rectangles of $P$ in $O(\lceil h/s \rceil \log n)$ passes using $O(s)$ space and $O(n \log^2 n)$ time for every $s = \Omega(\log n)$. Given randomness, the pass complexity can be reduced to $O(h/s + \log n)$ w.h.p.

**Approximate Convex Hulls.** Given the hardness result shown in Theorem 4, we know that one cannot have a constant-pass streaming algorithm that uses $o(h)$ space to compute the convex hull. In view of this, to have constant-pass $o(h)$-space streaming algorithms, one may consider computing an approximate convex hulls. There are several results studying on how to efficiently find an approximate convex hull in the streaming model, based on a given error measurement. The error criterion varies from the Euclidean distance [24], and Hausdorff metric distance [29,30], to the relative area error [35]. These algorithms use a single pass, $O(s)$ space, and can bound the given error measurement by a function of $s$.

**Paper Organization.** In Section 2, we present a new $O(n \log h)$-time RAM algorithm to compute the convex hull. Then, in Section 3, we present a constant-pass streaming algorithm. In Section 4, we present two W-stream algorithms, both of which use $O(s)$ space where $s$ can be as small as $O(\log n)$. Finally, in Section 5, we generalize the previous lower bound result.
Table 1 Categorization of four $O(n \log h)$-time algorithms for convex hull.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>any $r \geq 1$</td>
<td>Chan and Chen 2007 [13]</td>
<td>This paper</td>
</tr>
</tbody>
</table>

2 Yet another $O(n \log h)$-time algorithm in the RAM model

Our streaming algorithm is based on a RAM algorithm, which we present in this section. This RAM algorithm is a modification of Kirkpatrick and Seidel’s ultimate convex hull algorithm in the RAM model [28]. Chan and Chen’s streaming algorithm [13] is also based on Kirkpatrick and Seidel’s algorithm, and thus the structure of these two streaming algorithms have some similarities. The changes are made so that our streaming algorithm does not have to rely on solving linear programs, thus reducing the computation cost compared to Chan and Chen’s algorithm.

In what follows, we only discuss how to compute the upper hull because the lower hull can be computed analogously. Formally, computing the upper hull $U(P)$ of a point set $P$ means outputting that part of the extreme points $v_1, v_2, \ldots, v_t \in \text{ext}(P)$ in clockwise order so that $v_1$ is the leftmost point in $P$ and $v_t$ is the rightmost point in $P$, tie-breaking by picking the point with the largest $y$-coordinate, so that all points in $P$ lie below or on the line passing through $v_i, v_{i+1}$ for each $1 \leq i < t$. Note that each of $v_1, v_2, \ldots, v_t$ has a unique $x$-coordinate, and each line that passes through $v_i$ and $v_{i+1}$ for $1 \leq i < t$ has a finite slope.

Roughly speaking, Kirkpatrick and Seidel’s ultimate convex hull algorithm [28] evenly divides the point set into two subsets by a vertical line $t : x = \mu$, finds the hull edge in the upper hull that crosses $t$, and recurses on the two separated subsets. By appealing to the point-line duality, finding the crossing hull edge is equivalent to solving a linear program. Chan and Chen’s streaming algorithm is adapted from this implementation of the ultimate convex hull algorithm. Their algorithm evenly divides the point set into $r + 1$ subsets for $r \geq 1$ by $r$ vertical lines, finds the hull edges in the upper hull that cross these vertical lines, and recurses on the $r + 1$ separated subsets. Finding these $r$ crossing hull edges is equivalent to solving $r$ linear programs, where the constraint sets for each are the same but the objective functions are different.

In [11, Section 2], Chan gives another version of Kirkpatrick and Seidel’s ultimate convex hull algorithm, that finds a suitable (possibly random) extreme point, divides the point set into two by $x$-coordinate, and recurses. The extreme point can be found by elementary techniques. Our streaming algorithm is adapted from the latter algorithm. It finds $r$ suitable extreme points for $r \geq 1$, divides the point set into $r + 1$ subsets by $x$-coordinate, and recurses on each subset. Though this generalization sounds straightforward, finding the $r$ suitable extreme points needs a different approach from that for finding a single suitable extreme point. We reduce finding these $r$ suitable extreme points to computing the upper hulls of $n/(r+1)$ small point sets. This reduction is the key observation of our RAM algorithm and is described in detail in the subsequent paragraphs. These four algorithms are categorized in Table 1.

Given $r$, our algorithm partitions $P$ arbitrarily into $G_1, G_2, \ldots, G_{n/(r+1)}$ so that each $G_j$ has size in $[1, r+1]$, and then computes the upper hull of each $G_j$. Let $Q$ be the union of the slopes of the hull edges in the upper hull of $G_1, G_2, \ldots, G_{n/(r+1)}$, which is a multiset. Let $\sigma_k$ be the slope of rank $k|Q|/(r+1)$ in $Q$, for $k \in [1, r]$, in other words, $\sigma_k$ is the $k$th ($r+1$)-quantile in $Q$. To simplify the presentation, let $\sigma_0 = -\infty$ and $\sigma_{r+1} = \infty$. Let $s_k$ be
the extreme point in $P$ that supports slope $\sigma_k$, for each $k \in [0, r + 1]$. That is, for every point $p \in P$ draw a line passing through $p$ with slope $\sigma_k$, and pick $s_k$ as the point whose line has the highest $y$-intercept. We define $s_0 = p_L$, the point with the smallest $x$-coordinate, and $s_{r+1} = p_R$, the point with the largest $x$-coordinate. If any $s_k$ has more than one candidates, pick the point that has the largest $y$-coordinate. Let $x(p)$ denotes the $x$-coordinate of point $p$, and let $\sigma(p,q)$ denote the slope of the line that passes through points $p$ and $q$.

We use these $s_1, s_2, \ldots, s_r$ as the $r$ suitable extreme points with which to refine $P$ into $P_1, P_2, \ldots, P_{r+1}$ where we say the $s_i$ are suitable in that each $P_k$ has size bounded by $O(|P|/(r+1))$. Initially, set $P_k = \emptyset$ for all $k \in [1, r+1]$. The refinement applies the cascade-pruning described in Lemma 7 on $G_j$ for each $j \in [1, n/(r+1)]$, which uses the known pruning technique stated in Lemma 6 as a building block, and works as follows:

- **Step 1.** Compute the extreme points $v_1, v_2, \ldots, v_t \in U(G_j)$ in clockwise order.
- **Step 2.** Set $P_k \leftarrow P_k \cup \{v_i : i \in [\alpha, \beta], x(s_{k-1}) < x(v_i) < x(s_k)\}$ for each $k \in [1, r + 1]$, where $v_\alpha$ (resp. $v_\beta$) is the extreme point in $G_j$ that supports $\sigma_{k-1}$ (resp. $\sigma_k$).

The pruning in Step 2 is two-fold. For any $i < \alpha$, if $x(v_i) \leq x(s_{k-1})$, then $v_i$ cannot be placed in $P_k$. Otherwise $x(v_i) > x(s_{k-1})$, Case 2 of Lemma 7 applies. Again, $v_i$ cannot be placed in $P_k$. Similarly, $v_i$ for any $i > \beta$ cannot be placed in $P_k$ either. Finally, remove the points that lie below or on the line passing through $s_{k-1}, s_k$ from $P_k$ for each $k \in [1, r+1]$.

**Lemma 6** (Chan, [11]). Given a point set $P \subset \mathbb{R}^2$ and a slope $\sigma$, let $s$ be the extreme point in $P$ that supports $\sigma$. Then, for any pair of points $p, q \in P$ where $x(p) < x(q)$,
- **Case 1.** If $\sigma(p,q) \leq \sigma$ and $x(q) \leq x(s)$, then $q \notin U(P)$.
- **Case 2.** If $\sigma(p,q) \geq \sigma$ and $x(p) \geq x(s)$, then $p \notin U(P)$.

**Lemma 7** (Cascade-pruning). Given a point set $P \subset \mathbb{R}^2$ and a slope $\sigma$, let $s$ be the extreme point in $P$ that supports $\sigma$. Then, for any $G \subseteq P$ whose $U(G) = \{v_1, v_2, \ldots, v_t\}$, $x(v_1) < x(v_2) < \cdots < x(v_t)$, and where $\delta \in [1, t]$ is such that $v_\delta$ is the extreme point in $G$ that supports $\sigma$, we have:
- **Case 1.** If $x(v_i) \leq x(s)$ for some $i \in [\delta + 1, t]$, then $v_{\delta+1}, \ldots, v_t \notin U(P)$.
- **Case 2.** If $x(v_i) \geq x(s)$ for some $i \in [1, \delta - 1]$, then $v_1, \ldots, v_{\delta-1} \notin U(P)$.

**Proof.** Observe that $\sigma(v_j, v_{j+1}) \geq \sigma$ for all $j \in [1, \delta - 1]$ and $\sigma(v_{j-1}, v_j) \leq \sigma$ for all $j \in [\delta + 1, t]$ because $v_1, v_2, \ldots, v_t$ are extreme points in $U(G)$ in clockwise order and $v_\delta$ is the extreme point in $G$ that supports $\sigma$. Since there is an $i \in [\delta + 1, t]$ such that $x(v_i) \leq x(s)$, we have $x(v_j) \leq x(s)$ for each $j \in [\delta + 1, t]$. The above are exactly the conditions of Case 1 in Lemma 6 for all point pairs $(v_{j-1}, v_j)$ whose $j \in [\delta + 1, t]$. Thus, $v_j \notin U(P)$ for all $j \in [\delta + 1, t]$. The other case can be proved analogously. △

We get the exact bound for each $P_k$ in Lemma 8, noting that $|P_k| \leq \frac{3}{4} |P|$ for $r = 1$.

**Lemma 8.** $|P_k| \leq \left(\frac{\delta}{r+1} - \frac{1}{(r+1)^2}\right) |P| \leq 2|P|/(r+1)$ for each $k \in [1, r+1]$.

**Proof.** To ensure that, for every $k \in [1, r+1]$, $P_k$ is a small fraction of $P$, we use the cascade-pruning procedure described in Lemma 7. Let $\{v_1, v_2, \ldots, v_t\} = U(G_j)$ for some $j \in [n/(r+1)]$ where $x(v_1) < x(v_2) < \cdots < x(v_t)$. Let $v_\alpha_j$ (resp. $v_\beta_j$) be the extreme point in $G_j$ that supports $\sigma_{k-1}$ (resp. $\sigma_k$).

Let $n_j$ be the number of points in $P_k \cap G_j$. Recall that $P_k$ does not contain any $v_i$ for any $i \notin [\alpha_j, \beta_j]$, and hence $n_j \leq \beta_j - \alpha_j + 1$. Observe that point pair $(v_1, v_{j+1})$ has slope in the open interval $(\sigma_{k-1}, \sigma_k)$ for each $i \in [\alpha_j, \beta_j - 1]$. Since $\sigma_{k-1}$ (resp. $\sigma_k$) is the
RAM Algorithm: Compute the upper hull $U(P)$ of $P$.

1. Let $G_1, G_2, \ldots, G_{n/(r+1)}$ be any partition of $P$ such that each $G_j$ has size in $[1, r+1]$;
2. $Q \leftarrow \emptyset$;
3. foreach $G_j$, in the partition do
   4. Compute the upper hull $v_1, v_2, \ldots, v_t$ of $G_j$;
   5. for $i = 1$ to $t-1$ do
      6. $\sigma \leftarrow$ the slope of the line passing through $v_i, v_{i+1}$;
      7. $Q \leftarrow Q \cup \{\sigma\}$;
   8. end
9. end
10. for $k = 1$ to $r$ do
11. $\sigma_k \leftarrow$ the $k|Q|/(r+1)$-th smallest slope in $Q$;
12. $s_k \leftarrow$ the extreme point in $P$ that supports $\sigma_k$;
13. end
14. $(s_0, \sigma_0, s_{r+1}, \sigma_{r+1}) \leftarrow (p_L, -\infty, p_R, \infty)$;
15. for $k = 1$ to $r+1$ do
16. $P_k \leftarrow \emptyset$;
17. foreach $G_j$, in the partition do
18. Compute the upper hull $v_1, v_2, \ldots, v_t$ of $G_j$;
19. Find the extreme point $v_\alpha$ (resp. $v_\beta$) in $G_j$ that supports $\sigma_{k-1}$ (resp. $\sigma_k$);
20. $P_k \leftarrow P_k \cup \{v_\alpha, v_{\alpha+1}, \ldots, v_\beta\}$;
21. end
22. Remove the points that lie below or on the line passing through $s_{k-1}, s_k$ from $P_k$;
23. if $P_k \neq \emptyset$ then
24. Recurse on $P_k \cup \{s_{k-1}, s_k\}$;
25. end
26. end

$(k-1)|Q|/(r+1)$-th largest slope (resp. the $k|Q|/(r+1)$-th largest slope) in $Q$, $Q$ has at most $|Q|/(r+1)$ slopes in the open interval $(\sigma_{k-1}, \sigma_k)$. This yields that

$$
\sum_{j=1}^{n/(r+1)} n_j - 1 \leq \frac{|Q|}{r+1} \Rightarrow \sum_{j=1}^{n/(r+1)} n_j \leq \frac{|Q|}{r+1} + \frac{n}{r+1} \leq \frac{r|P|}{(r+1)^2} + \frac{|P|}{r+1}
$$

The last inequality holds because $|Q| \leq r|P|/(r+1)$, and it establishes that the number of points from all $G_j$’s that comprise $P_k$ for each $k \in [1, r+1]$ is at most $2|P|/(r+1)$. \hfill \triangleleft

For each $k \in [1, r+1]$, if $P_k \neq \emptyset$, then our algorithm recurses on $P_k \cup \{s_{k-1}, s_k\}$. This ensures that every subproblem has an input that contains some intermediate extreme point(s), i.e. not the leftmost and rightmost extreme points, and any two subproblems where one is not an ancestor or a descendant of the other have an empty intersection in their intermediate extreme point set. As a result,

\begin{lemma}
Our algorithm has $O(h)$ leaf subproblems.
\end{lemma}

Here we analyze the running time of the RAM algorithm for the case of $r = O(1)$ and defer the discussion for the case of $r = \omega(1)$ until the section on streaming algorithms. Let $T_C$ be the recursive computation tree of the RAM algorithm. The root of $T_C$ represents the initial problem of the recursive computation. Every node in $T_C$ has at most $r+1$ child nodes, each of which represents a recursive subproblem.
For a computation node with the input point set $P$ whose $|P| < r$, we use any $O(|P| \log r)$-time algorithm to compute the convex hull. Otherwise, we need to compute $|P|/(r + 1)$ convex hulls of point sets of size at most $r + 1$, which runs in $O(|P| \log r)$ time (Lines 1-9). In addition, the quantile selection in $Q$ has the running time $O(|Q| \log r) = O(|P| \log r)$ (Line 11). The $r$ suitable extreme points can be found in $O(|P| \log r)$ time by Lemma 15 (Line 12). The pruning procedure can be done in $O(|P| \log r)$ time by a simple merge (Lines 15-26). Hence, each computation node needs $O(|P| \log r)$ time.

Since each child subproblem has an input set $P_k \cup \{s_{k-1}, s_k\}$ of size at most $2|P|/(r + 1) + 2$ (Lemma 8), the running time of child subproblem is an $(2/(r + 1))$-fraction of its parent subproblem. Hence, $T_C$ is an $(2/(r + 1))$-fading computation tree where Edelsbrunner and Shi [17] define a recursive computation tree to be $\alpha$-fading for some $\alpha < 1$ if the running time of a child subproblem is an $\alpha$-fraction of its parent. In [11], Chan extends Edelsbrunner and Shi’s results and obtains that, if an $\alpha$-fading recursive computation tree has $L$ leaf nodes and the total running time of the nodes on each level is at most $F$, then the recursive computation tree has total running time $O(F \log L)$. Our algorithm has $O(h)$ leave nodes (Lemma 9) and $O(|P| \log r)$ time for the computation nodes on each level because two subproblems on the same level have their inputs only intersected at one of their extreme points. We get:

**Theorem 10.** The RAM algorithm runs in $O(n \log h \log \log h)$ time, and for $r = O(1)$ it is an $O(n \log h)$-time algorithm.

## 3 A Simpler and Faster Streaming Algorithm

In this section, we show how to adapt our RAM algorithm to the streaming model. Our streaming algorithm is the same as our RAM algorithm, but we execute the subproblems on $T_C$ in BFS order. That is, starting from the root of $T_C$, all subproblems on $T_C$ of the same level are solved together in a round, then their invoked subproblems are solved together in the next round, and so on. We will see in a moment that our algorithm needs to scan the input $O(1)$ times for each round. Therefore, to have an $O(1)$-pass streaming algorithm, our approach requires $r = n^\delta$ for some positive constant $\delta < 1$. By setting $r = n^\delta$, we have:

**Lemma 11.** By setting the parameter $r$ to be $n^\delta$ for any constant $\delta \in (0,1)$, the recursive computation tree $T_C$ has $O(\delta^{-1}h)$ nodes.

**Proof.** This lemma holds because $T_C$ has depth $O(\log_\delta n) = O(\delta^{-1})$ by Lemma 8 and $T_C$ has $O(h)$ leaf nodes by Lemma 9.

We assign a unique identifier $z \in [1,|T_C|]$ to each of $|T_C| = O(\delta^{-1}h)$ subproblems. Let $S_z$ be the subproblem on $T_C$ whose identifier is $z$. For each $z \in [1,|T_C|]$, $S_z$ has input point set $P_z$. $P_z$ is a subsequence of $P$ and is given to $S_z$ as an input stream of $|P_z|$ points. Our algorithm will generate $P_z$ more than once for $S_z$ to access, for all $z \in [1,|T_C|]$. The data structures used in $S_z$ also are suffixed with $z$. To compute $S_z$, naively we need $O(|P_z|)$ space. We will see in a moment that given $P_z$, how to solve $S_z$ using $O(r \log |P_z|)$ space in $O(r \log |P_z| + |P_z| \log r)$ time. We will also see how to generate the input for all the subproblems on $T_C$ of depth $d > 0$ in $O(1)$ passes. We now establish all these claims, after which we will be ready to prove Theorem 1. We decompose $S_z$ into the following three subtasks and describe the algorithms for the subtasks in the subsequent subsections.

1. Given $P_z$, obtain the $r$ quantile slopes $\sigma_1, \sigma_2, \ldots, \sigma_r$.
2. Given $P_z$ and $\sigma_1, \sigma_2, \ldots, \sigma_r$, obtain the $r$ suitable extreme points $s_1, s_2, \ldots, s_r$.
3. After the ancestor subproblems of $S_z$ (excluding $S_z$) are all solved, generate $P_z$. 

3.1 Obtaining the \( r \) quantile slopes

To find the \( r \) quantile slopes for \( S_z \) (Lines 1-11 in the RAM algorithm) using small space, we use a Greenwald and Khanna [21] quantile summary structure, abbreviated as \( QS_z \). This summary is a data structure that supports two operations: insert a slope \( QS_z.\text{insert}(\sigma) \) and query for (an estimate of) the \( t \)-th smallest slope \( QS_z.\text{query}(t) \) in \( Q_z \). Given access to \( QS_z \), we do not have to store the entire \( P_z \) to obtain the \( r \) quantile slopes. Instead, we invoke \( QS_z.\text{insert}(\sigma) \) for each slope \( \sigma \in Q_z \). After updating all slopes in \( Q_z \), we obtain an estimate of the \( (r+1) \)-quantile of \( Q_z \) by invoking \( QS_z.\text{query}(k\lfloor Q_z\lfloor/(r+1)) \) for all \( k \in [1,r] \).

\[ QS_z.\text{query}(k\lfloor Q_z\lfloor/(r+1)) \]

returns an estimate \( \hat{\sigma}_k \) that has an additive error \( c|Q_z| \) in the rank, where \( c \) is a parameter to be determined. We set \( c = \varepsilon/(r+1) \) for some constant \( \varepsilon > 0 \) so that the additive error cannot increase the depth of \( T_C \) by more than a constant factor. Precisely, because the obtained \( \hat{\sigma}_k \) has the rank in the range \([k-\varepsilon](Q_z\lfloor/(r+1)),(k+\varepsilon](Q_z\lfloor/(r+1)]\) for each \( k \in [1,r] \), we need to replace Lemma 8 with Corollary 12. Such a replacement increases the depth of \( T_C \) from \( O(\log r n) = O(\delta^{-1}) \) to \( O(\log r/(1+\varepsilon) n) = O(\delta^{-1}) + o(1) \).

\[ \text{Corollary 12. } |P_k| \leq \left(\frac{22 c}{r+1} - \frac{1}{r+1}\right)|P| \leq 2(1+\varepsilon)|P|/(r+1) \text{ for each } k \in [1,r+1]. \]

The summary \( QS_z \) needs \( O\left(\frac{1}{\varepsilon} \log(c|Q_z|)\right) \) space, and therefore the space usage for each subproblem is \( O((r/\varepsilon) \log((\varepsilon/\varepsilon)|Q_z|)) \). In [39], it shows that Greenwald and Khanna’s quantile summary needs \( O(\log |Q_z|) \) time for an update and \( O(\log r + \log \log(|Q_z|/r)) \) for a query. Because \( S_z \) conducts \( O(r) \) updates and \( O(r) \) queries, we get:

\[ \text{Lemma 13. Given } P_z, \text{ some streaming algorithm can find the } r \text{ approximate quantile slopes in } Q_z \text{ to within any } O(1) \text{ factor in } O(r \log(|P_z| + r)) \text{ time using } O(r \log(|P_z|/r)) \text{ space.} \]

3.2 Obtaining the \( r \) suitable extreme points

To find the \( r \) suitable extreme points in \( P_z \) (Line 12 in the RAM algorithm), a naive implementation, which would update the supporting points of \( \hat{\sigma}_k \) for all \( k \in [1,r] \) once for each point \( p \in P_z \), needs \( O(r|P_z|) \) running time. To reduce the running time to the claimed time complexity \( O(\log |P_z| + |P_z| \log r) \), we need the following observation.

\[ \text{Observation 14. For any non-singleton set } G \text{ whose extreme points in the upper hull } U(G) \text{ from left to right are } v_1, v_2, \ldots, v_t, \text{ the point in } G \text{ that supports a given slope } \sigma \text{ is} \]

\[ s = \begin{cases} v_1 & \text{if } \sigma > \sigma(v_1,v_2) \\ v_2 & \text{if } \sigma < \sigma(v_{t-1},v_t) \\ v_i & \text{if } \sigma(v_{t-1},v_t) \geq \sigma \geq \sigma(v_i,v_{i+1}) \text{ for some } i \in [2,t-2] \end{cases} \]

To find the extreme points in \( P_z \) that supports \( \hat{\sigma}_k \) for all \( k \in [1,r] \), we compute the extreme points \( v_1, v_2, \ldots, v_t \) in \( P_z \) from left to right, generate a (sorted) list \( \ell_A \) of slopes \( \sigma(v_1,v_2), \sigma(v_2,v_3), \ldots, \sigma(v_{t-1},v_t) \), and merge \( \ell_A \) with another (sorted) list \( \ell_B \) of the approximate \((r+1)\)-quantile slopes \( \hat{\sigma}_1, \hat{\sigma}_2, \ldots, \hat{\sigma}_r \). By Observation 14, the point \( \hat{s}_k \) in \( P_z \) that supports \( \hat{\sigma}_k \) for each \( k \in [1,r] \) can be easily determined by the its predecessor and successor in \( \ell_A \). Scanning the merged list suffices to get \( \hat{s}_1, \hat{s}_2, \ldots, \hat{s}_k \). Though the above reduces the time complexity to \( O(r + |P_z| \log |P_z|) \), the space complexity \( O(|P_z|) \) is much higher than the claimed space complexity \( O(r \log r|P_z|) \) for \( r \ll |P_z| \). To remedy, again, we reduce this problem to computing the upper hulls of \(|P_z|/(r+1)\) smaller point sets. First, we partition \( P_z \) arbitrarily into \([G_1,G_2,\ldots,G_{r+1}] \) so that each group \( G_i \) has size \(|G_i| \in [1,r+1] \) points. Then, for each \( G_i \) we apply the above accordingly. We get:

\[ \text{Lemma 15. Given } P_z \text{ and sorted } \sigma_1, \ldots, \sigma_r, \text{ some streaming algorithm can find the extreme points in } P_z \text{ that support } \sigma_i \text{ for all } i \in [1,r] \text{ in } O(r + |P_z| \log r) \text{ time using } O(r) \text{ space.} \]
3.3 Generating the input point set $P_z$ for each subproblem $S_z$

Recall that we execute the subproblems in $T_C$ in BFS order. Upon executing the subproblems of depth $d$ for any $d > 0$, all the subproblems of depth $< d$ are done and the associated $r$ quantile slopes and $r$ suitable extreme points are memoized in memory. For $d = 0$, we need to generate the input for the initial problem $S_0$, i.e. $P$, so scanning over $P$ suffices.

Given the associated $r$ quantile slopes and $r$ suitable extreme points for all the subproblems of depth less than $d$, to generate the input point sets for all the subproblems of depth $d$, we can directly execute Lines 15-26 in the RAM algorithm for all the subproblems of depth less than $d$ and ignore Lines 1-14 because the intermediate values, the quantile slopes and suitable extreme points, are already computed and kept in memory. Initially, we allocate a buffer $B_z$ of size $r+1$ for each subproblem $S_z$ of depth less than $d$ so as to temporarily store the incoming input points, i.e. points in $P_z$. Then, we scan $P$ on the input tape once and for each input point $p$ in $P$, we place $p$ in the buffer $B_o$ of $S_o$. Once any buffer $B_z$ gets full or the input terminates, we let $B_z$ be some $G_i$, a part in the partition of $P_z$, and apply the pruning procedure stated in Lines 15-26 in the RAM algorithm. Those points that survive the pruning are flushed, one by one, into the buffers of $S_z$’s child subproblems. We apply the above iteratively until we reach the end of the input tape. The space usage counted on each $S_z$ is $O(|B_z|) = O(r)$ and the overall running time to generate the input point set for all the subproblems of depth $d > 0$ is $O(dn \log r)$ because all the subproblems of each depth $i \in [1, d-1)$ computes the upper hull of points sets, disjoint subsets of $P$. Hence, we get:

Lemma 16. Some streaming algorithm can generate the input for all depth-$d$ subproblems on $T_C$ for each $d \in [0, \text{depth}(T_C)]$ using $O(1)$ passes, $O(hr)$ space, and $O(dn \log r)$ time.

Proof of Theorem 1. For $r = n^\delta$, $T_C$ has $O(\delta^{-1}h)$ nodes and depth $O(\delta^{-1})$ by Lemma 11, 8. Hence, the space complexity of our streaming algorithm is the sum of $O(\delta^{-1}h)$ times the space complexity in Lemma 13, 15, and $O(\delta^{-1})$ times the space complexity in Lemma 16. The overall space complexity is $O(\delta^{-1}hn^\delta \log n)$. One can obtain the space bound $O(\min(\delta^{-1}hn^\delta \log n, n))$ by checking whether $hn^\delta \log n > n$ before proceeding to the subproblems on the next depth, where $h$ is the number of subproblems executed so far and thus $h = O(\delta^{-1}h)$. If so, we compute the convex hull by a RAM algorithm. Analogously, we have that the pass (resp. time) complexity of our streaming algorithm is $O(\delta^{-1})$ (resp. $O(\delta^{-2}n \log n)$).

4 A W-Stream Algorithm Of Nearly-Optimal Pass-Space Tradeoff

Demetrescu et al. [15] establish a general scheme to convert PRAM algorithms to W-stream algorithms. Theorem 17 is an implication of their main result.

Theorem 17 (Demetrescu et al. [15]). If there exists a PRAM algorithm that uses $m$ processors to compute the convex hull of $n$ given points in $t$ rounds, then there exists an $O(s)$-space $O(mt/s)$-pass W-stream algorithm to compute the convex hull.

There is a long line of research that studies how to compute the convex hull of $n$ given points efficiently in parallel [1,3,4,14,19,23]. In particular, Akl’s PRAM algorithm [1] uses $O(n^2)$ processors and runs in $O(n^{1-\varepsilon} \log h)$ time for any $\varepsilon \in (0, 1)$. Converting Akl’s PRAM algorithm to a W-stream algorithm by Theorem 17, we have:

Corollary 18. There exists an $O((n/s) \log h)$-pass W-stream algorithm that can compute the convex hull of $n$ given points using $O(s)$ space.
The optimal work, i.e. the total number of primitive operations that the processors perform, for any parallel algorithm in the algebraic decision tree model to compute the convex hull is $O(n \log h)$ [23, 28]. Therefore the W-stream algorithm stated in Corollary 18 is already the best possible among those W-stream algorithms that are converted from a PRAM algorithm in the algebraic decision tree model by Theorem 17. However, in this Section, we will show that such a tradeoff between pass complexity and space usage is suboptimal by devising a W-stream algorithm that has a better pass-space tradeoff. Together with the results shown in Section 5, we have that the pass-space tradeoff of our W-stream algorithm is nearly optimal.

### 4.1 Deterministic W-stream Algorithm

Our deterministic W-stream algorithm is the same as our streaming algorithm, except for the following differences:

- We set $r = 1$ (rather than $r = n^d$) for our deterministic W-stream algorithm. Thus, by Corollary 12 $\text{depth}(T_C)$ increases from $O(\delta^{-1})$ to $O(\log n)$, but the space usage of subproblem $S_z$ decreases from $O(n^d \log n)$ to $O(\log n)$ for each $z \in [1, |T_C|]$. Moreover, if the extreme point in the input $P$ that supports the approximate median slope is the leftmost point $p_L$ or the rightmost point $p_R$, i.e. the degenerate case, we replace it with the extreme point that supports $\sigma(p_L, p_R)$. In this way, each subproblem on $T_C$ has a unique extreme point and therefore the number of subproblems on $T_C$ is $O(h)$.

- Our streaming algorithm executes the subproblems on $T_C$ in BFS order, that is, all subproblems of depth $d$ are executed in a round for each $d \in [0, \text{depth}(T_C)]$. In contrast, our deterministic W-stream algorithm refines a single round into subrounds, in each of which it takes care of $O(s/\log n)$ subproblems, so as to bound the working space by $O(s)$.

- Note that algorithms in the W-stream model are capable of modifying the input tape. Formally, while scanning the input tape in the $i$-th pass, algorithms can write something on a write-only output stream; in the $(i + 1)$-th pass, the input tape read by algorithms is the output tape written in the $i$-th pass. Hence, our deterministic W-stream algorithm is able to assign an attribute to each point $p \in P$ to indicate that $p$ is an input of a certain subproblem. Moreover, our deterministic W-stream algorithm can write down the parameters for every subproblem on the output tape. In each subround, our deterministic W-stream algorithm needs to scan the input twice. The first pass is used to load the parameters of subproblems to be solved in the current subround. The second pass is used to scan the input tape and process the points that are the input points for the subproblems to be solved in the current subround.

**Proof of Theorem 2.** Suppose there are $h_d$ subproblems of depth $d$ on $T_C$ for each $d \in [0, \text{depth}(T_C)]$, then our deterministic W-stream algorithm has to execute $\sum_d \left[ \frac{h_d}{s/\log n} \right] = O((h/s) \log n)$ subrounds for any $s = \Omega(\log n)$. Because our deterministic W-stream algorithm scans the input tape twice for each subround, the pass complexity is $O((h/s) \log n)$.

As shown in Section 3, subproblem $S_z$ needs $O(|P_z| \log |P_z|)$ running time. Since the input of subproblems of depth $d$ on $T_C$ are disjoint subsets of $P$, for each $d \in [0, |T_C|]$. We get that the time complexity is $O(n \log^2 n).$ □

### 4.2 Randomized W-stream Algorithm

Observe that for $r = 1$, finding the $r$ approximate quantile slopes in $Q_z$ is exactly finding the approximate median slope in $Q_z$. Our algorithms mentioned previously all use Greenwald and Khanna quantile summary structure, which needs $O(\log n)$ space. In our randomized
W-stream algorithm, we replace the Greenwald and Khanna quantile summary with a random slope in $Q_2$, thereby reducing the space usage to $O(1)$. As noted by Bhattacharya and Sen [7], such a replacement cannot increase the depth of $T_C$ by more than a constant factor w.h.p.

**Proof of Theorem 3.** Similar to the arguments used in the proof of Theorem 2, the pass complexity of our randomized W-stream algorithm is \( \sum_{d \in [0, \text{depth}(T_C)]} \frac{h_d}{s/\Theta(1)} = O(h/s + \log n) \) for any $s = \Omega(1)$ w.h.p. and the time complexity is $O(n \log^2 n)$ w.h.p. ▲

5 Unconditional Lower Bound

In this section, we will show that any streaming (or W-stream) algorithm that can compute the convex hull in the streaming (and W-stream) model. Set disjointness is defined as follows. Alice has a private

\[
\mathcal{A} = \{a_1, a_2, \ldots, a_k\},
\]

\[
\mathcal{B} = \{b_1, b_2, \ldots, b_k\},
\]

\[
\mathcal{C} = \{c_1, c_2, \ldots, c_k\},
\]

\[
\mathcal{D} = \{d_1, d_2, \ldots, d_k\},
\]

... and only if

\[
\mathcal{A} \cap \mathcal{B} = \emptyset.
\]

We note here that the lower bound holds even if the output is $\text{ext}(P)$, rather than the set $\text{ext}(P)$.

We construct a point set $U$ so that it is hard to compute the convex hull of point set $P = Q \cup \{(1,0), (-1,0)\}$ for all $Q \subseteq U$. Let $C_1, C_2$ be concentric half circles. The radius of $C_1$ equals 1 and that of $C_2$ is any value in $(k, 1)$ for some $k$ to be determined later. Let $a_0, a_1, \ldots, a_{n+1}$ be points distributed evenly on $C_1$ so that $a_0 = (1,0)$ and $a_{n+1} = (-1,0)$. Define $b_0, b_1, \ldots, b_{n+1}$ on $C_2$ similarly. Let $k$ be the distance between the origin $O$ and the line $\overline{a_i a_{i+2}}$ for any $i \in [0, n-1]$. Let $U$ be the set \( \{a_i : i \in [1, n]\} \cup \{b_i : i \in [1, n]\} \).

We need the following geometric property of points in $U$ for the hardness proof.

**Lemma 19.** For every $Q \subseteq U$, let $R = \text{ext}(Q \cup \{(1,0), (-1,0)\})$. We have that (1) $a_i \in Q \Rightarrow a_i \in R$, and (2) $(b_i \in Q \Rightarrow b_i \in R)$ s.t. $a_i \notin Q$.

**Proof.** Due to space constraints, we defer the proof to the full version of this paper [18]. ▲

Lemma 19 implies the fact that, for every $Q \subseteq U$, $|\text{ext}(Q \cup \{(1,0), (-1,0)\})| = |Q| + 2$ if and only if $a_i$ and $b_i$ are not both contained in $Q$ for each $i$. Given this fact, we are ready to perform a reduction from the set disjointness problem (a two-party communication game) to computing the convex hull in the streaming (and W-stream) model. Set disjointness is defined as follows. Alice has a private $(\alpha n)$-size subset $A$ of $[n]$, and Bob has another private $(\alpha n)$-size subset $B$ of $[n]$ for some constant $\alpha < 1/2$. The goal is to answer whether $A$ and $B$ have an non-empty intersection. Based on the hardness result of set-disjointness, due to Kalyanasundaram and Schintger [27], we are ready to prove Theorem 4.

**Proof of Theorem 4.** Due to space constraints, we defer the proof to the full version of this paper [18]. ▲

**References**


47:12 Streaming Algorithms for Planar Convex Hulls


Deterministic Treasure Hunt in the Plane with Angular Hints

Sébastien Bouchard
Sorbonne Université, CNRS, INRIA, LIP6, F-75005 Paris, France
sebastien.bouchard@lip6.fr

Yoann Dieudonné
Laboratoire MIS, Université de Picardie Jules Verne, Amiens, France
yoann.dieudonne@u-picardie.fr

Andrzej Pelc
Département d’informatique, Université du Québec en Outaouais, Gatineau, Canada
pelc@uqo.ca

Franck Petit
Sorbonne Université, CNRS, INRIA, LIP6, F-75005 Paris, France
franck.petit@lip6.fr

Abstract

A mobile agent equipped with a compass and a measure of length has to find an inert treasure in
the Euclidean plane. Both the agent and the treasure are modeled as points. In the beginning,
the agent is at a distance at most $D > 0$ from the treasure, but knows neither the distance nor any
bound on it. Finding the treasure means getting at distance at most 1 from it. The agent makes
a series of moves. Each of them consists in moving straight in a chosen direction at a chosen
distance. In the beginning and after each move the agent gets a hint consisting of a positive
angle smaller than $2\pi$ whose vertex is at the current position of the agent and within which the
treasure is contained. We investigate the problem of how these hints permit the agent to lower
the cost of finding the treasure, using a deterministic algorithm, where the cost is the worst-case
total length of the agent’s trajectory. It is well known that without any hint the optimal (worst
case) cost is $\Theta(D^2)$. We show that if all angles given as hints are at most $\pi$, then the cost can
be lowered to $O(D)$, which is optimal. If all angles are at most $\beta$, where $\beta < 2\pi$ is a constant
unknown to the agent, then the cost is at most $O(D^{2-\epsilon})$, for some $\epsilon > 0$. For both these positive
results we present deterministic algorithms achieving the above costs. Finally, if angles given as
hints can be arbitrary, smaller than $2\pi$, then we show that cost $\Theta(D^2)$ cannot be beaten.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms,
Computing methodologies → Mobile agents

Keywords and phrases treasure hunt, deterministic algorithm, mobile agent, hint, plane

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.48
1 Introduction

Motivation. A tourist visiting an unknown town wants to find her way to the train station or a skier lost on a slope wants to get back to the hotel. Luckily, there are many people that can help. However, often they are not sure of the exact direction: when asked about it, they make a vague gesture with the arm swinging around the direction to the target, accompanying the hint with the words “somewhere there”. In fact, they show an angle containing the target. Can such vague hints help the lost traveller to find the way to the target? The aim of the present paper is to answer this question.

The model and problem formulation. A mobile agent equipped with a compass and a measure of length has to find an inert treasure in the Euclidean plane. Both the agent and the treasure are modeled as points. In the beginning, the agent is at a distance at most $D > 0$ from the treasure, but knows neither the distance nor any bound on it. Finding the treasure means getting at distance at most 1 from it. In applications, from such a distance the treasure can be seen. The agent makes a series of moves. Each of them consists in moving straight in a chosen direction at a chosen distance. In the beginning and after each move the agent gets a hint consisting of a positive angle smaller than $2\pi$ whose vertex is at the current position of the agent and within which the treasure is contained. We investigate the problem of how these hints permit the agent to lower the cost of finding the treasure, using a deterministic algorithm, where the cost is the worst-case total length of the agent’s trajectory. It is well known that the optimal cost of treasure hunt without hints is $\Theta(D^2)$. (The algorithm of cost $O(D^2)$ is to trace a spiral with jump 1 starting at the initial position of the agent, and the lower bound $\Omega(D^2)$ follows from Proposition 9 which establishes this lower bound even assuming arbitrarily large angles smaller than $2\pi$ given as hints.)

Our results. We show that if all angles given as hints are at most $\pi$, then the cost of treasure hunt can be lowered to $O(D)$, which is optimal. Our real challenge here is in the fact that hints can be angles of size exactly $\pi$, in which case the design of a trajectory always leading to the treasure, while being cost-efficient in terms of traveled distance, is far from obvious.

If all angles are at most $\beta$, where $\beta < 2\pi$ is a constant unknown to the agent, then we prove that the cost is at most $O(D^{2-\epsilon})$, for some $\epsilon > 0$. Finally, we show that arbitrary angles smaller than $2\pi$ given as hints cannot be of significant help: using such hints the cost $\Theta(D^2)$ cannot be beaten.

For both our positive results we present deterministic algorithms achieving the above costs. Both algorithms work in phases “assuming” that the treasure is contained in increasing squares centered at the initial position of the agent. The common principle behind both algorithms is to move the agent to strategically chosen points in the current square, depending on previously obtained hints, and sometimes perform exhaustive search of small rectangles from these points, in order to guarantee that the treasure is not there. This is done in such a way that, in a given phase, obtained hints together with small rectangles exhaustively searched, eliminate a sufficient area of the square assumed in the phase to eventually permit finding the treasure.

In both algorithms, the points to which the agent travels and where it gets hints are chosen in a natural way, although very differently in each of the algorithms. The main difficulty is to prove that the distance travelled by the agent is within the promised cost. In the case of the first algorithm, it is possible to cheaply exclude large areas not containing the
treasure, and thus find the treasure asymptotically optimally. For the second algorithm, the agent eliminates smaller areas at each time, due to less precise hints, and thus finding the treasure costs more.

Due to lack of space, the details of one of the algorithms and proofs of several results are omitted and will appear in the journal version of the paper.

Related work. The problem of treasure hunt, i.e., searching for an inert target by one or more mobile agents was investigated under many different scenarios. The environment where the treasure is hidden may be a graph or a plane, and the search may be deterministic or randomized. An early paper [4] showed that the best competitive ratio for deterministic treasure hunt on a line is 9. In [8] the authors generalized this problem, considering a model where, in addition to travel length, the cost includes a payment for every turn of the agent. The book [2] surveys both the search for a fixed target and the related rendezvous problem, where the target and the finder are both mobile and their role is symmetric: they both cooperate to meet. This book is concerned mostly with randomized search strategies. Randomized treasure hunt strategies for star search, where the target is on one of $m$ rays, are considered in [13]. In [17, 19] the authors study relations between the problems of treasure hunt and rendezvous in graphs. The authors of [3] study the task of finding a fixed point on the line and in the grid, and initiate the study of the task of searching for an unknown line in the plane. This research is continued, e.g., in [12, 15]. In [18] the authors concentrate on game-theoretic aspects of the situation where multiple selfish pursuers compete to find a target, e.g., in a ring. The main result of [14] is an optimal algorithm to sweep a plane in order to locate an unknown fixed target, where locating means to get the agent originating at point $O$ to a point $P$ such that the target is in the segment $OP$. In [10] the authors consider the generalization of the search problem in the plane to the case of several searchers. Collective treasure hunt in the grid by several agents with bounded memory is investigated in [9, 16]. In [5], treasure hunt with randomly faulty hints is considered in tree networks. By contrast, the survey [7] and the book [6] consider pursuit-evasion games, mostly on graphs, where pursuers try to catch a fugitive target trying to escape.

2 Preliminaries

Since for $D \leq 1$ treasure hunt is solved immediately, in the sequel we assume $D > 1$. Since the agent has a compass, it can establish an orthogonal coordinate system with point $O$ with coordinates $(0,0)$ at its starting position, the $x$-axis going East-West and the $y$-axis going North-South. Lines parallel to the $x$-axis will be called horizontal, and lines parallel to the $y$-axis will be called vertical. When the agent at a current point $a$ decides to go to a previously computed point $b$ (using a straight line), we describe this move simply as “Go to $b$”. A hint given to the agent currently located at point $a$ is formally described as an ordered pair $(P_1, P_2)$ of half-lines originating at $a$ such that the angle clockwise from $P_1$ to $P_2$ contains the treasure. The line containing points $A$ and $B$ is denoted by $(AB)$. A segment with extremities $A$ and $B$ is denoted by $[AB]$ and its length is denoted $|AB|$. Throughout the paper, a polygon is defined as a closed polygon (i.e., together with the boundary). For a polygon $S$, we will denote by $B(S)$ (resp. $I(S)$) the boundary of $S$ (resp. the interior of $S$, i.e., the set $S \setminus B(S)$). A rectangle is defined as a non-degenerate rectangle, i.e., with all sides of strictly positive length. A rectangle with vertices $A,B,C,D$ (in clockwise order) is denoted simply by $ABCD$. A rectangle is straight if one of its sides is vertical.
Algorithm 1 Procedure RectangleScan(R).

1: if \( k \) is odd then
2:   for \( i = 0 \) to \( k - 1 \) step 2 do
3:     Go to \( a_i \); Go to \( b_i \);
4:     Go to \( b_{i+1} \); Go to \( a_{i+1} \)
5:   end for
6: else
7:   for \( i = 0 \) to \( k - 2 \) step 2 do
8:     Go to \( a_i \); Go to \( b_i \);
9:     Go to \( b_{i+1} \); Go to \( a_{i+1} \)
10: end for
11: end if
12: Go to \( a_k \); Go to \( b_k \)
13: Go to \( a \)
14: end if

In our algorithms we use the following procedure \texttt{RectangleScan}(\( R \)) whose aim is to traverse a closed rectangle \( R \) (composed of the boundary and interior) with known coordinates, so that the agent initially situated at some point of \( R \) gets at distance at most 1 from every point of it and returns to the starting point. We describe the procedure for a straight rectangle whose vertical side is not shorter than the horizontal side. The modification of the procedure for arbitrarily positioned rectangles is straightforward. Let the vertices of the rectangle \( R \) be \( A, B, C \) and \( D \), where \( A \) is the North-West vertex and the others are listed clockwise. Let \( a \) be the point at which the agent starts the procedure.

The idea of the procedure is to go to vertex \( A \), then make a snake-like movement in which consecutive vertical segments are separated by a distance 1, and then go back to point \( a \). The agent ignores all hints gotten during the execution of the procedure. Suppose that the horizontal side of \( R \) has length \( m \) and the vertical side has length \( n \), with \( n \geq m \). Let \( k = \lfloor m \rfloor \). Let \( a_0, a_1, \ldots, a_k \) be points on the North horizontal side of the rectangle, such that \( a_0 = A \) and the distance between consecutive points is 1. Let \( b_0, b_1, \ldots, b_k \) be points on the South horizontal side of the rectangle, such that \( b_0 = D \) and the distance between consecutive points is 1.

The pseudocode of procedure \texttt{RectangleScan}(\( R \)) is given in Algorithm 1.

\begin{prop}
For every point \( p \) of the rectangle \( R \), the agent is at distance at most 1 from \( p \) at some time of the execution of Procedure \texttt{RectangleScan}(\( R \)). The cost of the procedure is at most \( 5n \cdot \max(m, 2) \), where \( n \geq m \) are the lengths of the sides of the rectangle.
\end{prop}

\section{Angles at most \( \pi \)}

In this section we consider the case when all angles given as hints are at most \( \pi \). Without loss of generality we can assume that they are all equal to \( \pi \), completing any smaller angle to \( \pi \) in an arbitrary way: this makes the situation even harder for the agent, as hints become less precise. For such hints we show Algorithm \texttt{TreasureHunt1} that finds the treasure at cost \( O(D) \). This is of course optimal, as the treasure can be at any point at distance at most \( D \) from the starting point of the agent.

For angles of size \( \pi \), every hint is in fact a half-plane whose boundary line \( L \) contains the current location of the agent. For simplicity, we will code such a hint as \((L, \text{right})\) or \((L, \text{left})\), whenever the line \( L \) is not horizontal, depending on whether the indicated half-plane is to
the right (i.e., East) or to the left (i.e., West) of \( L \). For any non-horizontal line \( L \) this is non-ambiguous. Likewise, when \( L \) is horizontal, we will code a hint as \((L, \text{up})\) or \((L, \text{down})\), depending on whether the indicated half-plane is up (i.e., North) from \( L \) or down (i.e., South) from \( L \).

In view of the work on \( \phi \)-self-approaching curves (cf. [1]) we first note that there is a big difference of difficulty between obtaining our result in the case when angles given as hints are strictly smaller than \( \pi \) and when they are at most \( \pi \), as we assume. A \( \phi \)-self-approaching curve is a planar oriented curve such that, for each point \( B \) on the curve, the rest of the curve lies inside a wedge of angle \( \phi \) with apex in \( B \). In [1], the authors prove the following property of these curves: for every \( \phi < \pi \) there exists a constant \( c(\phi) \) such that the length of any \( \phi \)-self-approaching curve is at most \( c(\phi) \) times the distance \( D \) between its endpoints. Hence, for angles \( \phi \) strictly smaller than \( \pi \), our result could possibly be derived from the existing literature: roughly speaking, the agent should follow a trajectory corresponding to any \( \phi \)-self-approaching curve to find the treasure at a cost linear in \( D \). Even then, transforming the continuous scenario of self-approaching curves to our discrete scenario presents some difficulties. However, the crucial problem is this: the result of [1] holds only when \( \phi < \pi \) (the authors also emphasize that for each \( \phi \geq \pi \), the property is false), and thus the above derivation is no longer possible for our purpose when \( \phi = \pi \). Actually, this is the real difficulty of our problem: handling angles equal to \( \pi \), i.e., half-planes.

We further observe that a rather straightforward treasure hunt algorithm of cost \( O(D \log D) \), for hints being angles of size \( \pi \), can be obtained using an immediate corollary of a theorem proven in [11] by Grünbaum: each line passing through the centroid of a convex polygon cuts the polygon into two convex polygons with areas differing by a factor of at most \( \frac{\pi}{2} \). Suppose for simplicity that \( D \) is known. Starting from the square of side length \( 2D \), centered at the initial position of the agent, this permits to reduce the search area from \( P \) to at most \( \frac{\pi D}{2} \) in a single move. Hence, after \( O(\log D) \) moves, the search area is small enough to be exhaustively searched by procedure \texttt{RectangleScan} at cost \( O(D) \). However, the cost of each move during the reduction is not under control and can be only bounded by a constant multiple of \( D \), thus giving the total cost bound \( O(D \log D) \). By contrast, our algorithm controls both the remaining search area and the cost incurred in each move, yielding the optimal cost \( O(D) \).

The high-level idea of our Algorithm \texttt{TreasureHunt1} is the following. The agent acts in phases \( j = 1, 2, 3, \ldots \) where in each phase \( j \) the agent “supposes” that the treasure is in a straight square \( R_j \) centered at the initial position of the agent, and of side length \( 2^j \). When executing a phase \( j \), the agent successively moves to distinct points with the aim of using the hints at these points to narrow the search area that initially corresponds to \( R_j \). In our algorithm, this narrowing is made in such a way that the remaining search area is always a straight rectangle. Often this straight rectangle is a strict superset of the intersection of all hints that the agent was given previously. This would seem to be a waste, as we are searching some areas that have been previously excluded. However, this loss is compensated by the ease of searching description and subsequent analysis of the algorithm, due to the fact that, at each stage, the search area is very regular.

During a phase, the agent proceeds to successive reductions of the search area by moving to distinct locations, until it obtains a rectangular search area that is small enough to be searched directly at low cost using procedure \texttt{RectangleScan}. In our algorithm, such a final execution of \texttt{RectangleScan} in a phase is triggered as soon as the rectangle has a side smaller than \( 4 \). If the treasure is not found by the end of this execution of procedure \texttt{RectangleScan}, the agent learns that the treasure cannot be in the supposed straight square \( R_j \) and starts
Figure 1 In Figure (a) the agent received a good hint \((L_1, \text{right})\) at the point \(p\) of a rectangular search area \(ABCD\). In Figure (b) it received a bad hint \((L_1, \text{right})\) at the point \(p\) and hence it moved to point \(p'\) and got a hint \((L_2, \text{left})\). In both figures the excluded half-planes are shaded.

the next phase from scratch by forgetting all previously received hints. This forgetting again simplifies subsequent analysis. The algorithm terminates at the latest by the end of phase \(j_0 = \lceil \log_2 D \rceil + 1\), in which the supposed straight square \(R_{j_0}\) is large enough to contain the treasure. Hence, if the cost of a phase \(j\) is linear in \(2^j\), then the cost of the overall solution is linear in the distance \(D\).

In order to give the reader deeper insights in the reasons why our solution is valid and has linear cost, we need to give more precise explanations on how the search area is reduced during a given phase \(j \geq 2\) (when \(j = 1\), the agent makes no reduction and directly scans the small search area using procedure \texttt{RectangleScan}). Suppose that in phase \(j \geq 2\) the agent is at the center \(p\) of a search area corresponding to a straight rectangle \(R\), every side of which has length between 4 and \(2^j\) (note that this is the case at the beginning of the phase), and denote by \(A, B, C\) and \(D\) the vertices of \(R\) starting from the top left corner and going clockwise. In order to reduce rectangle \(R\), the agent uses the hint at point \(p\). The obtained hint denoted by \((L_1, x_1)\) can be of two types: either a \textit{good} hint or a \textit{bad} hint. A good hint is a hint whose line \(L_1\) divides one of the sides of \(R\) into two segments such that the length \(y\) of the smaller one is at least \(1\). A bad hint is a hint that is not good.

If the received hint \((L_1, x_1)\) is good, then the agent narrows the search area to a rectangle \(R' \subset R\) having the following three properties:
1. \(R \setminus R'\) does not contain the treasure.
2. The difference between the perimeters of \(R\) and \(R'\) is \(2y \geq 2\).
3. The distance from \(p\) to the center of \(R'\) is exactly \(\frac{y}{2}\).

and then moves to the center of \(R'\).

An illustration of such a reduction is depicted in Figure 1(a). The reduced search area \(R'\) is the rectangle \(ABde\).

If the agent receives a bad hint, say \((L_1, \text{right})\), at the center of a rectangular search area \(R\), we cannot apply the same method as the one used for a good hint: this is the reason for the distinction between good and bad hints. If we applied the same method as before, we
could obtain a rectangular search area \( R' \) such that the difference between the perimeters of \( R \) and \( R' \) is at least \( 2y \). However, in the context of a bad hint, the difference \( 2y \) may be very small (even null), and hence there is no significant reduction of the search area. In order to tackle this problem, when getting a bad hint at the center \( p \) of \( R \), the agent moves to another point \( p' \) which is situated in the half-plane \((L_1, \text{right})\) at distance 2 from \( p \), perpendicularly to \( L_1 \). This point \( p' \) is chosen in such a way that, regardless of what is the second hint, we can ensure that two important properties described below are satisfied.

The first property is that by combining the two hints, the agent can decrease the search area to a rectangle \( R' \subset R \) whose perimeter is smaller by \( 2 \) compared to the perimeter of \( R \), as it is the case for a good hint, and such that \( R \setminus R' \) does not contain the treasure. This decrease follows either directly from the pair of hints, or indirectly after having scanned some relatively small rectangles using procedure \texttt{RectangleScan}. In the example depicted in Fig. 1 (b), after getting the second hint \((L_2, \text{left})\), the agent executes procedure \texttt{RectangleScan}(ss'd'd) followed by \texttt{RectangleScan}(gg'h'h) and moves to the center of the new search area \( R' \) that is the rectangle \( Agpm \). Note that the part of \( R' \) not excluded by the two hints and by the procedure \texttt{RectangleScan} executed in rectangles \( ss'd'd \) and \( gg'h'h \) is only the small quadrilateral bounded by line \( L_2 \) and the segments \([AB], [s'd'] \) and \([gh] \). However, in order to preserve the homogeneity of the process, we consider the entire new search area \( R' \) which is a straight rectangle whose perimeter is smaller by at least \( 2 \), compared to that from \( R \). This follows from the fact that no side of \( R \) has length smaller than \( 4 \). The agent finally moves to the center of \( R' \).

The second property is that all of this (i.e., the move from \( p \) to \( p' \), the possible scans of small rectangles and finally the move to the center of \( R' \)) is done at a cost linear in the difference of perimeters of \( R \) and \( R' \). The two properties together ensure that, even with bad hints, the agent manages to reduce the search area in a significant way and at a small cost. So, regardless of whether hints are good or not, we can show that the cost of phase \( j \) is in \( \mathcal{O}(2^j) \) and the treasure is found during this phase if the initial square is large enough. The difficulty of the solution is in showing that the moves prescribed by our algorithm in the case of bad hints guarantee the two above properties, and thus ensure the correctness of the algorithm and the cost linear in \( D \).

\begin{theorem}
Consider an agent \( A \) and a treasure located at distance at most \( D \) from the initial position of \( A \). By executing Algorithm \texttt{TreasureHunt1}, agent \( A \) finds the treasure after having traveled a distance \( \mathcal{O}(D) \).
\end{theorem}

\section{Angles bounded by \( \beta < 2\pi \)}

In this section we consider the case when all hints are angles upper-bounded by some constant \( \beta < 2\pi \), unknown to the agent. The main result of this section is Algorithm \texttt{TreasureHunt2} whose cost is at most \( \mathcal{O}(D^{2-\epsilon}) \), for some \( \epsilon > 0 \). For a hint \((P_1, P_2)\) we denote by \((\overline{P}_1, \overline{P}_2)\) the complement of \((P_1, P_2)\).

\subsection{High level idea}

In Algorithm \texttt{TreasureHunt2}, similarly as in the previous algorithm, the agent acts in phases \( j = 1, 2, 3, \ldots \), where in each phase \( j \) the agent “supposes” that the treasure is in the straight square centered at its initial position and of side length \( 2^j \). The intended goal is to search each supposed square at relatively low cost, and to ensure the discovery of the treasure by the time the agent finishes the first phase for which the initial supposed square contains the
Deterministic Treasure Hunt in the Plane with Angular Hints

However, the similarity with the previous solution ends there: indeed, the hints that may now be less precise do not allow us to use the same strategy within a given phase. Hence we adopt a different approach that we outline below and that uses the following notion of tiling. Given a square $S$ with side of length $x > 0$, $Tiling(i)$ of $S$, for any non-negative integer $i$, is the partition of square $S$ into $4^i$ squares with side of length $\frac{x}{2^i}$. Each of these squares, called tiles, is closed, i.e., contains its border, and hence neighboring tiles overlap in the common border.

Let us consider a simpler situation in which the angle of every hint $(P_1, P_2)$ is always equal to the bound $\beta$: the general case, when the angles may vary while being at most $\beta$, adds a level of technical complexity that is unnecessary to understand the intuition. In the considered situation, the angle of each excluded zone $(P_1, P_2)$ is always the same as well. The following property holds in this case: there exists an integer $i_\beta$ such that for every square $S$ and every hint $(P_1, P_2)$ given at the center of $S$, at least one tile of $Tiling(i_\beta)$ of $S$ belongs to the excluded zone $(P_1, P_2)$.

In phase $j$, the agent performs $k$ steps: we will indicate later how the value of $k$ should be chosen. At the beginning of the phase, the entire square $S$ is white. In the first step, the agent gets a hint $(P_1, P_2)$ at the center of $S$. By the above property, we know that $(P_1, P_2)$ contains at least one tile of $Tiling(i_\beta)$ of $S$, and we have the guarantee that such a tile cannot contain the treasure. All points of all tiles included in $(P_1, P_2)$ are painted black in the first step. This operation does not require any move, as painting is performed in the memory of the agent. As a result, at the end of the first step, each tile of $Tiling(i_\beta)$ of $S$ is either black or white, in the following precise sense: a black tile is a tile all of whose points are black, and a white tile is a tile all of whose interior points are white.

In the second step, the agent repeats the painting procedure at a finer level. More precisely, the agent moves to the center of each white tile $t$ of $Tiling(i_\beta)$ of $S$. When it gets a hint at the center of a white tile $t$, there is at least one tile of $Tiling(i_\beta)$ of $t$ that can be excluded. As in the first step, all points of these excluded tiles are painted black. Note that a tile of $Tiling(i_\beta)$ of $t$ is actually a tile of $Tiling(2i_\beta)$ of $S$. Moreover, each tile of $Tiling(i_\beta)$ of $S$ is made of exactly $4^{i_\beta}$ tiles of $Tiling(2i_\beta)$ of $S$. Hence, as depicted in Figure 2, the property we obtain at the end of the second step is as follows: each tile of $Tiling(2i_\beta)$ of $S$ is either black or white.

In the next steps, the agent applies a similar process at increasingly finer levels of tiling. More precisely, in step $2 < s \leq k$, the agent moves to the center of each white tile of $Tiling((s - 1)i_\beta)$ of $S$ and gets a hint that allows it to paint black at least one tile of $Tiling(si_\beta)$ of $S$. At the end of step $s$, each tile of $Tiling(si_\beta)$ of $S$ is either black or white. We can show that at each step $s$ the agent paints black at least $\frac{1}{4^{i_\beta}}$th of the area of $S$ that is white at the beginning of step $s$.

After step $k$, each tile of $Tiling(k \cdot i_\beta)$ of $S$ is either black or white. These steps permit the agent to exclude some area without having to search it directly, while keeping some regularity of the shape of the black area. The agent paints black a smaller area than excluded by the hints but a more regular one. This regularity enables to turn the next process in the area remaining white. Indeed, the agent subsequently executes a brute-force searching that consists in moving to each white tile of $Tiling(k \cdot i_\beta)$ of $S$ in order to scan it using the procedure RectangleScan. If, after having scanned all the remaining white tiles, it has not found the treasure, the agent repaints white all the square $S$ and enters the next phase. Thus we have the guarantee that the agent finds the treasure by the end of phase $\lceil \log_2 D \rceil + 1$, i.e., a phase in which the initial supposed square is large enough to contain the treasure. The question is: how much do we have to pay for all of this? In fact, the cost depends on the
value that is assigned to $k$ in each phase $j$. The value of $k$ must be large enough so that the distance travelled by the agent during the brute-force searching is relatively small. At the same time, this value must be small enough so that the distance travelled during the $k$ steps is not too large. A good trade-off can be reached when $k = \lceil \log_4 i_\beta \sqrt{2} j \rceil$. Indeed, as highlighted in the proof of correctness, it is due to this carefully chosen value of $k$ that we can beat the cost $\Theta(D^2)$ necessary without hints, and get a complexity of $O(D^{2-\epsilon})$, where $\epsilon$ is a positive real depending on $i_\beta$, and hence depending on the angle $\beta$.

4.2 Algorithm and analysis

In this subsection we describe our algorithm in detail, prove its correctness and analyze its complexity. We can prove there exists a function $\text{index} : (0, 2\pi) \rightarrow \mathbb{N}^+$ that has the following properties, for any angle $0 < \alpha < 2\pi$.

1. For every square $S$ and for every hint $(P_1, P_2)$ of size $2\pi - \alpha$ obtained at the center of $S$, there exists a tile of $\text{Tiling}(\text{index}(\alpha))$ of $S$ included in $(P_1, P_2)$.
2. For every angle $\alpha' < \alpha$, we have $\text{index}(\alpha) \leq \text{index}(\alpha')$.

In the sequel, the integer $\text{index}(\alpha)$ is called the index of $\alpha$. Algorithm 2 gives a pseudo-code of the main algorithm of this section. It uses the function $\text{Mosaic}$ described in Algorithm 3 that is the key technical tool permitting the agent to reduce its search area. The agent interrupts the execution of Algorithm 2 as soon as it gets at distance 1 from the treasure, at which point it can “see” it and thus treasure hunt stops.

In the following, a square is called black if all its points are black. A square is called white if all points of its interior are white. (In a white square, some points of its border may be black).

Lemma 3. For any positive integers $i$ and $k$, consider an agent executing function $\text{Mosaic}(i,k)$ from its initial position $O$. Let $S$ be the straight square centered at $O$ with side of length $2^i$. For every positive integer $j \leq \lceil \log_4 i_\beta \sqrt{2} \rceil$, at the end of the $j$-th execution of the first loop (lines 5 to 20) in $\text{Mosaic}(i,k)$, each tile of $\text{Tiling}(jk)$ of $S$ is either black or white.

Figure 2 White and black tiles at the end of the first and the second step of a phase, for square $S = ABCD$ and $i_\beta = 2$. 

(a) At the end of a first step for a hint $(P_1, P_2)$.
(b) At the end of a second step.
Algorithm 2 TreasureHunt2.

1: IndexNew := 1
2: i := 1
3: loop
4: repeat
5: IndexOld := IndexNew
6: IndexNew := Mosaic(i, IndexOld)
7: until IndexNew = IndexOld
8: i := i + 1
9: end loop

Lemma 4. For every positive integers i and k, a call to function Mosaic(i,k) has cost at most $2^{\frac{3+\log_4(4^k-1)}{2}}+2k+8$.

Let $\psi$ be the index of $2\pi - \beta$. The next proposition follows from the aforementioned properties of the function index.

Proposition 5. Let $(P_1, P_2)$ be any hint. The index of $(P_1, P_2)$ is at most $\psi$.

Using Lemmas 3, 4 and Proposition 5 we prove the final result of this section.

Theorem 6. Consider an agent A and a treasure located at distance at most $D$ from the initial position of $A$. By executing Algorithm TreasureHunt2, agent A finds the treasure after having traveled a distance in $O(D^{2-\epsilon})$, for some $\epsilon > 0$.

Proof. We will use the following two claims.

Claim 7. Let $i \geq 1$ be an integer. The number of executions of the repeat loop in the $i$-th execution of the external loop in Algorithm 2 is bounded by $\psi$.

Claim 8. The distance traveled by the agent before variable $i$ becomes equal to $\lceil \log_2 D \rceil + 2$ in the execution of Algorithm 2 is $O(D^{2-\epsilon})$, where $\epsilon = \frac{1}{2}(1 - \log_4(4^\psi - 1)) > 0$.

Proof of the claim. In view of the fact that the returned value of every call to function Mosaic in the execution of Algorithm 2 is at most $\psi$, it follows that in each call to function Mosaic($*, k$) the parameter $k$ is always at most $\psi$. Hence, in view of Claim 7 and Lemma 4, as long as variable $i$ does not reach the value $\lceil \log_2 D \rceil + 2$, the agent traveled a distance at most

$$\psi \cdot \sum_{i=1}^{\lceil \log_2 D \rceil + 1} 2^{\frac{3+\log_4(4^\psi - 1)}{2}}+2\psi+8$$

(1)

$$\leq \psi \cdot 2^{\log_2 D + 1} \cdot 2^{\frac{3+\log_4(4^\psi - 1)}{2}}+2\psi+9$$

(2)

$$\leq \psi \cdot 2^{2\psi+12+\log_4(4^\psi - 1)} \cdot (\frac{3+\log_4(4^\psi - 1)}{2})$$

(3)

$$= \psi \cdot 2^{2\psi+12+\log_4(4^\psi - 1)} \cdot (\frac{3+\log_4(4^\psi - 1)}{2})$$

(4)

By (4), the total distance traveled by the agent executing Algorithm 2 is $O(D^{2-\epsilon})$ where $\epsilon = \frac{1}{2}(1 - \log_4(4^\psi - 1))$. Since $\psi$ is a positive integer, we have $0 < \log_4(4^\psi - 1) < 1$ and hence $\epsilon > 0$. This ends the proof of the claim.

$\blacksquare$
Algorithm 3 Function $\text{Mosaic}(i,k)$.

1: $O:= \text{the initial position of the agent}$
2: $S:= \text{the straight square centered at } O \text{ with sides of length } 2^i$
3: Paint white all points of $S$
4: $\text{IndexMax}:=k$
5: for $j = 1$ to $\lfloor \log_4 \sqrt{2i} \rfloor$ do
6:   for all tiles $t$ of $\text{Tiling}((j-1)k)$ of $S$ do
7:     if $t$ is white then
8:       Go to the center of $t$
9:       Let $(P_1, P_2)$ be the obtained hint
10:      $k':= \text{index of } (P_1, P_2)$
11:     if $k' > \text{IndexMax}$ then
12:        $\text{IndexMax}:=k'$
13:    end if
14:   if $\text{IndexMax} = k$ then
15:     for all tiles $t'$ of $\text{Tiling}(k)$ of $t$ such that $t' \subset (P_1, P_2)$ do
16:       Paint black all points of $t'$
17:     end for
18:   end if
19: end if
20: end for
21: end for
22: if $\text{IndexMax} = k$ then
23:   for all tiles $t$ of $\text{Tiling}(k(\lfloor \log_4 \sqrt{2i} \rfloor))$ of $S$ do
24:     if $t$ is white then
25:       Go to the center of $t$
26:       Execute $\text{RectangleScan}(t)$
27:     end if
28:   end for
29: end if
30: Go to $O$
31: return $\text{IndexMax}$

Assume that the theorem is false. As long as variable $i$ does not reach $\lfloor \log_2 D \rfloor + 2$, the agent cannot find the treasure, as this would contradict Claim 8. Thus, in view of Claim 7, before the time $\tau$ when variable $i$ reaches $\lfloor \log_2 D \rfloor + 2$ the treasure is not found. By Algorithm 2, this implies that during the last call to function $\text{Mosaic}$ before time $\tau$, the function returns a value that is equal to its second input parameter. This implies that during this call, the agent has executed lines 23 to 28 of Algorithm 3: more precisely, there is some integer $x$ such that from each white tile $t$ of $\text{Tiling}(x)$ of the straight square $S$ that is centered at the initial position of the agent and that has sides of length $2^i$, the agent has executed function $\text{RectangleScan}(t)$. Hence, at the end of the execution of lines 23 to 28, the agent has seen all points of each white tile of $\text{Tiling}(x)$ of $S$. Moreover, in view of Lemma 3, we know that the tiles that are not white, in $\text{Tiling}(x)$ of $S$, are necessarily black. Given a black tile $\sigma$ of $\text{Tiling}(x)$, each point of $\sigma$ is black, which, in view of lines 15 to 17 of Algorithm 3, implies that $\sigma$ cannot contain the treasure. Since square $S$ necessarily contains the treasure, it follows that the agent must find the treasure by the end of the last
5 Arbitrary angles

We finally observe that if hints can be arbitrary angles smaller than $2\pi$ then the treasure hunt cost $\Theta(D^2)$ cannot be improved in the worst case.

\textbf{Proposition 9.} If hints can be arbitrary angles smaller than $2\pi$ then the optimal cost of treasure hunt for a treasure at distance at most $D$ from the starting point of the agent is $\Omega(D^2)$.

6 Conclusion

For hints that are angles at most $\pi$ we gave a treasure hunt algorithm with optimal cost linear in $D$. For larger angles we showed a separation between the case where angles are bounded away from $2\pi$, when we designed an algorithm with cost strictly subquadratic in $D$, and the case where angles have arbitrary values smaller than $2\pi$, when we showed a quadratic lower bound on the cost. The optimal cost of treasure hunt with large angles bounded away from $2\pi$ remains open. In particular, the following questions seem intriguing. Is the optimal cost linear in $D$ in this case, or is it possible to prove a super-linear lower bound on it? Does the order of magnitude of this optimal cost depend on the bound $\pi < \beta < 2\pi$ on the angles given as hints?

References


Competitive Searching for a Line on a Line Arrangement

Quirijn Bouts
ASML Veldhoven, the Netherlands

Thom Castermans¹
TU Eindhoven, the Netherlands
t.h.a.castermans@tue.nl

Arthur van Goethem
TU Eindhoven, the Netherlands
a.i.v.goethem@tue.nl

Marc van Kreveld²
Utrecht University, the Netherlands
m.j.vankreveld@uu.nl

Wouter Meulemans³
TU Eindhoven, the Netherlands
w.meulemans@tue.nl

Abstract
We discuss the problem of searching for an unknown line on a known or unknown line arrangement by a searcher S, and show that a search strategy exists that finds the line competitively, that is, with detour factor at most a constant when compared to the situation where S has all knowledge. In the case where S knows all lines but not which one is sought, the strategy is $79$-competitive. We also show that it may be necessary to travel on $\Omega(n)$ lines to realize a constant competitive ratio. In the case where initially, S does not know any line, but learns about the ones it encounters during the search, we give a $414.2$-competitive search strategy.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Competitive searching, line arrangement, detour factor, search strategy

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.49

Acknowledgements This work was started during the 2nd AGA Workshop, in Jan./Feb. 2017. The authors thank two anonymous reviewers for their extensive and detailed comments.

1 Introduction

Given a set $L$ of $n$ lines $\ell_0, \ell_1, \ldots, \ell_{n-1}$ in the plane, consider the arrangement $\mathcal{A}$ that they form as a geometric graph. Technically, $\mathcal{A}$ is not a graph due to half-infinite edges, but in our problem we can end each line at its extreme intersection points, and hence we can use the term graph without complications. We consider paths on $\mathcal{A}$. The cost of a path on $\mathcal{A}$ is the Euclidean length of that path. The distance between two points on $\mathcal{A}$ is the cost (or length) of the shortest path that stays on $\mathcal{A}$ between those points.

---

1 The Netherlands Organisation for Scientific Research (NWO) supports T.C. under project no. 314.99.117.
2 Supported by the Netherlands Organisation for Scientific Research on grant no. 612.001.651.
3 Supported by the Netherlands eScience Center (NLeSC) on project 027.015.G02.
Assume that a searcher $S$ is located on some vertex or edge of the graph. Denote its initial position by $O$. The searcher $S$ can only travel on the arrangement and is hence restricted to paths on $A$. Searcher $S$ is looking for a target line $\ell_t \in L$, but does not know which of the lines in $L$ corresponds with $\ell_t$. The searcher $S$ will recognize $\ell_t$ when it reaches any point on $\ell_t$ (necessarily at an intersection point with another line). We call this special line the target line, and assume that $O$ does not lie on $\ell_t$. If it would, the problem would be solved immediately. We consider two versions of the problem: one where $S$ knows the lines in $L$ and therefore $A$ completely, and one where $S$ only knows about the existence and parameters of a line once it reaches some point on it.

We will show that a search strategy exists by which $S$ can reach the target line competitively in both versions. In other words, $S$ can reach the target line with a detour factor bounded by a constant, when compared to the shortest path on $A$ to the target line. Competitive analysis is commonly used to compare “the cost of not knowing” with “the cost of knowing”. The maximum detour factor of a search strategy is known as its competitive ratio. The competitive ratio of a search problem is the infimum of the competitive ratios of all search strategies that solve that search problem.

The best known search problem is perhaps the one-dimensional problem of finding a point on a line from a starting position. If we know the distance $d$, but not whether it is to the left or to the right, the optimal strategy is to go left over a distance $d$ and then right over a distance $2d$. We find the point with competitive ratio $3$, which is optimal. If we don’t know the distance but we do know some (very) small lower bound $\epsilon$ on the distance, it is best to go $\epsilon$ to the left, then back and another $2\epsilon$ to the right, then back and another $4\epsilon$ to the left, and so on. This doubling strategy gives a competitive ratio of $9$, which is known to be optimal as proved by Beck and Newman [4] in 1970, see also [2, 13].

The problem of searching for a line in the plane without obstacles was studied by Baeza-Yates et al. [2] in various settings. The settings refer to the knowledge we have of the line, which can be its slope, its distance, both, or neither. If the slope of the line is known, the problem reduces to the one-dimensional problem just discussed. If only the distance is known, the optimal competitive ratio is $6.39...$. The problem of searching for a line a given distance away was posed by Bellman [5] in 1956 and solved by Isbell [20] in 1957. It is a classic in recreational mathematics and often posed as a swimmer in the fog, trying to reach the (straight) shore which is a known unit distance away, while swimming the least in the worst case. If the slope nor the distance of a line to be found is known, the best known competitive ratio is $13.81...$, which is realized by a logarithmic spiral search strategy.

Competitive analysis of algorithms was introduced by Sleator and Tarjan for analyzing the list update problem [24]. Here the lack of knowledge is the next online requests. In geometric situations, the lack of knowledge is often the environment itself or the location of something to be found (by seeing or reaching it). The main motivation of such problems comes from the navigation of robots in unknown environments. More generally, searching for a target in environments where either the target or the environment is unknown is a basic problem, and competitive analysis is a fundamental way to understand what is in principle possible in such exploration problems. We list a few main results on searching and competitive analysis in geometric and geometric-graph environments; for an extensive overview see also [15]. We begin by noting that there is no $c$-competitive search strategy to find an unknown target node in a known graph, for example when the graph is a star.

When searching for an unknown target on a line, but additional information on the distance to the target is known, alternative results can be obtained [8, 17]. Demaine et al. [13] show that searching for an unknown target on a line with cost depending on both
search distance and turns can be done competitively with cost $9OPT + 2d$, where $d$ is the cost of one turn. Searching on multiple rays is studied in various papers [8, 13, 16, 23]; Kao et al. [22] give an optimal randomized algorithm. In yet other variants one can search with multiple searchers [3, 16].

Kalyanasundaram and Pruhs [21] consider visibility-based searching for a recognizable point in an unknown scene with convex obstacles. Their result on the competitive factor is not constant, but depends on the number of obstacles and their aspect ratio. Blum et al. [6] investigate similar problems for different classes of obstacles. Hoffmann et al. [18] show that an unknown simple polygon can be discovered completely with a competitive ratio of 26.5. There are various other visibility-based search problems addressed with competitive analysis (e.g.,[14, 19]).

A different setting where competitive strategies are investigated is routing in geometric graphs. Here an unknown geometric graph is given along with a source and target with known coordinates. We route a package from source to target over the nodes, but learn about the existence and coordinates of a node when we are at a neighbor. For triangulations, no $c$-competitive strategy exists, but for special triangulations like Delaunay and certain other geometric graphs, a constant competitive strategy does exist [7, 9, 11, 12]. Searching for an unknown target on a planar straight line graph with discovery based on Pokémon Go was investigated with competitive analysis recently [25].

Contributions. In Section 2 we give a preliminary result where we use only two lines and obtain a competitive ratio depending on their angle. Moreover, we show that, if we want to obtain a constant competitive ratio that does not depend on parameters of the arrangement, then the search strategy must allow for traversing at least half the lines in an arrangement. In Section 3 we describe and analyze such a strategy and show that this leads to a 79-competitive strategy. This is an upper bound on the relative cost of not knowing which line is sought. (Note that for slightly more complex objects like half-lines, no constant-competitive strategy exists by mimicking a star graph, see Figure 1.) In Section 4 we generalize the problem to the situation where the searcher does not know all lines beforehand. They learn about the existence of a line and its parameters only when the line is reached. We show that in this case a search strategy exists with competitive ratio 414.2. This is an upper bound on the relative cost of not knowing the lines at all.

Although our search problems and competitive ratios are new, the existing literature implies lower bounds for our versions. When all lines are known, we have a lower bound of 9, because the problem is at least as hard as the one-dimensional problem of finding a point on a line. Moreover, it is essentially also at least as hard as finding a fully unknown line in the plane, because we could be given a very dense set of lines where all movement is approximately possible and every line could be the target. The best known competitive ratio is 13.81... to find an unknown line, but this is not known to be optimal so it does not provide a true lower bound. In case we do not know the lines of the arrangement at all, we inherit the lower bound of searching on four rays (half-lines) for a point, which is 19.96... [1, 13]. The
line arrangement consists of two perpendicular lines, we start on their intersection, and we must explore. If we do not follow the optimal strategy for four rays, the target line was just out of reach at the place where we went less far, and perpendicular to that ray. With more than four rays, lines will intersect more than one ray and the argument no longer works.

2 Competitive searching on an arrangement

As a warm-up, assume that $S$ starts at the intersection of two lines $\ell_1$ and $\ell_2$ whose smaller intersection angle is $\alpha \leq \pi/2$. Furthermore, $S$ only traverses $\ell_1$ and $\ell_2$, disregarding all other lines for traversal.

▶ Theorem 1. The target line can be found with competitive ratio at most $29/\sin(\alpha/2)$.

Proof. Denote the starting point by $O$ and the target line by $\ell_t$. As a lower bound for reaching $\ell_t$ we use the Euclidean distance between $O$ and $\ell_t$, denoted by $x$, because a line $\ell_3$ through $O$ and normal to $\ell_t$ could exist.

Note that $\ell_t$ must intersect at least one of $\ell_1$ and $\ell_2$. Let $y$ be the distance on $\ell_1$ or $\ell_2$ to the closest intersection point $u$ of $\ell_t$ with $\ell_1$ and/or $\ell_2$. Since $\alpha$ is the smaller angle, the worst case occurs when the target line $\ell_t$ spans a triangle with the two initial lines $\ell_1$ and $\ell_2$ with an angle of $\pi - \alpha$; the worst ratio between $x$ and $y$ occurs when this triangle is equilateral with apex $O$. This is illustrated in Figure 2. By elementary geometry, we then have $y \leq x/\sin(\alpha/2)$.

The strategy to find $\ell_t$ is as follows. Let $d$ be the distance between $O$ and the vertex $v$ on $\ell_1$ or $\ell_2$ closest to it. First, $S$ travels to $v$ and back to $O$. Then $S$ travels the same distance $d$ in each of the other three directions on $\ell_1$ and $\ell_2$, and back to $O$ each time. After that we double $d$ and repeat. $S$ has achieved its goal when it reaches $u$, and therefore $\ell_t$.

We can view the traversal of $S$ on $\ell_1$ and $\ell_2$ as the traversal on four half-lines induced by $O$. One of these half-lines crosses $\ell_t$ at distance $y$. This is, by definition, where $u$ is. By the doubling strategy, $S$ will have traversed a total distance less than $5y$ on the half-line with $u$. On each of the other half-lines, $S$ has traversed at most a distance of $8y$. Summing up yields that the searcher travelled at most a distance of $29y$; using $y \leq x/\sin(\alpha/2)$, we find that the competitive ratio, bounded by $29y/x$, gives the claimed bound of $29/\sin(\alpha/2)$. ◀

We note that a tighter analysis of the same strategy will give a slightly better competitive ratio, and a different strategy where we traverse the half-lines over different distances will also give a better competitive ratio. However the strategy is not $c$-competitive for any constant $c$, since $\alpha$ can be arbitrarily small. Moreover, since this is a special case of the problem, we explore this strategy no further.

Below, we show that for any constant $c$, any $c$-competitive strategy must traverse $\Omega(n)$ lines. So the strategy of the proof of Theorem 1 cannot work, not even with the usage of some carefully chosen additional lines besides $\ell_1$ and $\ell_2$. 

Figure 2 Sketch of worst case.
Thus if we do not use line $\ell_i$ on the positive side of $\ell_{i+1}$, the construction will be such that the part of $\ell_i$ between its intersection with $h_i$ and its intersection with $\ell_{i-1}$ must be used by $S$ to reach $h_i$ with detour no more than $c$, because even the intersection of $\ell_{i-1}$ with $h_i$ has an $x$-coordinate that is too high.

In more detail, we construct the lines incrementally from $m$ down to 1, in pairs $\ell_i$ and then $h_i$, see Figure 3. We start with $\ell_m : y = x - 2$ and $h_m : y = 1$. Assume $\ell_{i+1}$ and $h_{i+1}$ are placed, and their intersection point $p_{i+1}$ is such that $d_{i+1} = \text{dist}(O, h_0 \cap \ell_{i+1}) > \text{dist}(h_0 \cap \ell_{i+1}, p_{i+1})$ (for $\ell_m$ and $h_m$ we made sure of this condition). Then we define $\ell_i$ by constructing two points on it. One is the point $(d_{i+1}/(2c), 0)$ on $h_0$; the other is the point on $h_{i+1}$ with $x$-coordinate $2cd_{i+1}$. This defines $\ell_i$. The line $h_i$ is chosen horizontal and low enough so that $\text{dist}(h_0 \cap \ell_i, p_i) < \text{dist}(O, h_0 \cap \ell_i) = d_{i+1}/(2c)$. Note that $d_m = 2$ and $d_i = 2/(2c)^m$.

To argue that this arrangement forces a searcher $S$ to walk on every line $\ell_i$ (and also $h_0$ where $S$ starts), we observe that we can reach the line $h_i$ in distance at most $d_{i+1}/c$ simply by following $h_0$ and $\ell_i$ only (we can do a little bit better but for the proof this is not needed). To reach $h_i$ $c$-competitively we must thus travel less than $d_{i+1}$ along $A$.

We cannot use line $\ell_{i+1}$ or any higher-indexed line, because all their vertices have $x$-coordinates at least $d_{i+1}$ so it must take $d_{i+1}$ or more to even reach $\ell_{i+1}$ or a later line. Thus if we do not use $\ell_i$, we must reach line $h_i$ on line $\ell_{i-1}$ or a lower-indexed line. By construction the intersection of $\ell_{i-1}$ and $h_i$ has $x$-coordinate $d_{i+1}$. Thus reaching $h_i$ from $\ell_{i-1}$ is not $c$-competitive. Furthermore, any line $\ell_j$ with $1 \leq j < i - 1$ must intersect $h_i$ right of the intersection with $\ell_{i-1}$ and thus for the same reason reaching $h_i$ via $\ell_j$ cannot be $c$-competitive.

In other words, we must use $\ell_i$ to get $c$-competitively to $h_i$, and any of the $m$ horizontal lines $h_1, \ldots, h_m$ can be the target line. Hence, a $c$-competitive strategy must visit and walk on each of $\ell_1, \ldots, \ell_m$. As $S$ starts on $h_0$, it thus walks on at least $m + 1 \geq n/2$ lines.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.pdf}
\caption{Placement of $\ell_i$ and $h_i$, given $\ell_{i+1}$ and $h_{i+1}$. Line $\ell_i$ is defined by the point with $x$-coordinate $d_{i+1}/(2c)$ on $h_0$ and the point with $x$-coordinate $2cd_{i+1}$ on $h_{i+1}$. Line $h_i$ is placed such that $\text{dist}(h_0 \cap \ell_{i+1}, p_i) < d_{i+1}/(2c)$.}
\end{figure}

\textbf{Theorem 2.} For any constant $c \geq 1$, there is an arrangement $\mathcal{A}$ of $n$ lines such that any $c$-competitive strategy must traverse at least $n/2 = \Omega(n)$ lines of $\mathcal{A}$ in the worst case.

\textbf{Proof.} We construct an arrangement $\mathcal{A}$ of $n = 2m + 1$ lines. The line $h_0$ is the $x$-axis, and searcher $S$ starts on $h_0$ at the origin $O$. Let $h_1, \ldots, h_m$ be horizontal lines that together with $h_0$ have a bottom-to-top order $h_0, h_1, \ldots, h_m$. Let $\ell_1, \ldots, \ell_m$ be $m$ lines with positive slope $\leq 1$, such that the upper envelope of $\ell_1, \ldots, \ell_m$ is a convex increasing function that contains all these lines in the same order. We ensure that these lines intersect $h_0$ on the positive side and in the order $\ell_1, \ldots, \ell_m$. The construction will be such that the part of $\ell_i$ between its intersection with $h_i$ and its intersection with $\ell_{i-1}$ must be used by $S$ to reach $h_i$ with detour no more than $c$, because even the intersection of $\ell_{i-1}$ with $h_i$ has an $x$-coordinate that is too high.

In more detail, we construct the lines incrementally from $m$ down to 1, in pairs $\ell_i$ and then $h_i$, see Figure 3. We start with $\ell_m : y = x - 2$ and $h_m : y = 1$. Assume $\ell_{i+1}$ and $h_{i+1}$ are placed, and their intersection point $p_{i+1}$ is such that $d_{i+1} = \text{dist}(O, h_0 \cap \ell_{i+1}) > \text{dist}(h_0 \cap \ell_{i+1}, p_{i+1})$ (for $\ell_m$ and $h_m$ we made sure of this condition). Then we define $\ell_i$ by constructing two points on it. One is the point $(d_{i+1}/(2c), 0)$ on $h_0$; the other is the point on $h_{i+1}$ with $x$-coordinate $2cd_{i+1}$. This defines $\ell_i$. The line $h_i$ is chosen horizontal and low enough so that $\text{dist}(h_0 \cap \ell_i, p_i) < \text{dist}(O, h_0 \cap \ell_i) = d_{i+1}/(2c)$. Note that $d_m = 2$ and $d_i = 2/(2c)^m$.

To argue that this arrangement forces a searcher $S$ to walk on every line $\ell_i$ (and also $h_0$ where $S$ starts), we observe that we can reach the line $h_i$ in distance at most $d_{i+1}/c$ simply by following $h_0$ and $\ell_i$ only (we can do a little bit better but for the proof this is not needed). To reach $h_i$ $c$-competitively we must thus travel less than $d_{i+1}$ along $A$.

We cannot use line $\ell_{i+1}$ or any higher-indexed line, because all their vertices have $x$-coordinates at least $d_{i+1}$ so it must take $d_{i+1}$ or more to even reach $\ell_{i+1}$ or a later line. Thus if we do not use $\ell_i$, we must reach line $h_i$ on line $\ell_{i-1}$ or a lower-indexed line. By construction the intersection of $\ell_{i-1}$ and $h_i$ has $x$-coordinate $d_{i+1}$. Thus reaching $h_i$ from $\ell_{i-1}$ is not $c$-competitive. Furthermore, any line $\ell_j$ with $1 \leq j < i - 1$ must intersect $h_i$ right of the intersection with $\ell_{i-1}$ and thus for the same reason reaching $h_i$ via $\ell_j$ cannot be $c$-competitive.

In other words, we must use $\ell_i$ to get $c$-competitively to $h_i$, and any of the $m$ horizontal lines $h_1, \ldots, h_m$ can be the target line. Hence, a $c$-competitive strategy must visit and walk on each of $\ell_1, \ldots, \ell_m$. As $S$ starts on $h_0$, it thus walks on at least $m + 1 \geq n/2$ lines. \hfill $\blacksquare$
A 3-competitive search strategy on a known arrangement

We continue with the general case where $S$ may start anywhere on any line and we make no assumptions on the angles between intersecting lines. For convenience we will assume the starting point to be at the origin $O$ and the line crossing through $O$ to be $ℓ_0$. If multiple lines cross the origin, we pick $ℓ_0$ to be the line that intersects any other line closest to $O$. We will assume $ℓ_0$ is horizontal. As the problem is rotation and translation invariant these assumptions do not change the problem. As before let $d$ be the distance to the closest intersection point on $ℓ_0$.

Consider the following search strategy for $S$. Searcher $S$ iteratively explores the arrangement starting from the origin. In iteration $i$ four paths of length $2^i \cdot d$ are explored starting at $O$. These paths are picked such that they maximize (minimize) the $x$- respectively $y$-coordinate that $S$ can achieve on the arrangement within distance $2^i \cdot d$ from $O$ (see Figure 4(a)). Specifically this results in the following strategy. First, $S$ traverses $ℓ_0$ over a distance $2d$ in the direction $+x$ and then returns back to $O$. Second, $S$ traverses $ℓ_0$ for a distance $2d$ in the direction $-x$ and back. Third, $S$ determines the point on $A$ with maximum $y$-coordinate it can reach when traversing over a distance $2d$; $S$ goes there and back. Symmetrically, $S$ also visits the point with lowest $y$-coordinate reachable within distance $2d$ from $O$. Upon returning to the origin the allowed distance is doubled and the process is repeated until $S$ finds $ℓ_1$.

Let $D_i$ be the distance travelled in iteration $i$. Let the points where $S$ ends when searching over a distance $D_i$ with minimum and maximum $x$- and $y$-coordinate be denoted $p_i^{-x}$, $p_i^{+x}$, $p_i^{-y}$, and $p_i^{+y}$, respectively. Figure 4(b) shows the four paths to $p_i^{+y}$, $p_i^{-y}$, $p_i^{-x}$, $p_i^{+x}$. Notice that the path for $D_{i+1}$ does not necessarily follow the path for $D_i$ as a less steep line may be followed to reach a steeper line sooner.

Lemma 3. The $y$-coordinate of $p_i^{+y}$ is at least twice that of $p_{i-1}^{+y}$. The symmetric statement holds for $p_i^{-y}$ and $p_{i-1}^{-y}$.

Proof. Observe that for any $p_{i-1}^{+y}$, the last line traversed on the path to $p_{i-1}^{+y}$ must have the steepest absolute slope. If not, we could get higher by staying on the line with steepest slope. When we traverse a distance $D_i$ instead of $D_{i-1}$, we have the option of staying on this steepest absolute slope line, and since $D_i = 2D_{i-1}$, we get at least twice as high just by staying on the line that contains $p_{i-1}^{+y}$.

Let $Q_i$ be the convex quadrilateral with the points $p_i^{-x}$, $p_i^{+x}$, $p_i^{-y}$, and $p_i^{+y}$ as vertices and let $R_i$ be the axis-parallel rectangle with these four points on its boundary (see Figure 5).
Trivially $Q_i \subset R_i$ and from Lemma 3 it immediately follows that $R_1 \subset R_2 \subset \ldots \subset R_k$.

▶ Lemma 4. $R_i \subset Q_{i+2}$.

Proof. Without loss of generality only consider the half-plane above $\ell_0$. We show that the triangle $p_i^{+x}p_{i+2}^{+x}p_{i+2}^{+y}$ contains the rectangle with bottom vertices $p_i^{+x}$ and $p_i^{+y}$ and top side through $p_{i+2}^{+y}$. We know that $p_i^{+x}p_{i+2}^{+y}$ is exactly four times the length of $p_i^{+x}p_i^{+y}$ as $\ell_0$ is horizontal. By Lemma 3 the $y$-coordinate of $p_{i+2}^{+y}$ is at least four times that of $p_i^{+y}$ (see Figure 6). By triangle inequality the $x$-coordinate of $p_{i+2}^{+y}$ must be between $p_i^{+x}$ and $p_{i+2}^{+x}$.

Let $x$ be the $x$-coordinate of $p_{i+2}^{+x}$, and $r = (-x, y)$ the vertex at the top-left corner of $R_i$. Consider the side $p_i^{+x}p_{i+2}^{+y}$ of the triangle and the line $p_{i+2}^{+y}r$. The slope of $p_{i+2}^{+y}r$ is $y/(3x)$. The slope of $p_i^{+x}p_{i+2}^{+y}$ depends on the exact location of $p_{i+2}^{+y}$. In the (impossible) worst case $p_{i+2}^{+y}$ is located at $(4x, 4y)$. Thus the slope of $p_i^{+x}p_{i+2}^{+y}$ is at least $y/(2x)$ and $r$ is below $p_i^{+x}p_{i+2}^{+y}$. Containment of $R_i$ in $Q_{i+2}$ trivially follows.

We observe that if the target line $\ell_i$ intersects $Q_i$ then $\ell_i$ will be found in iteration $i$ or before. Hence the distance travelled by the searcher is upper-bounded by the distance travelled up to and including iteration $i$. Suppose the searcher $S$ finds the target line $\ell_i$ in iteration $k$. We will use the rectangle $R_{k-3}$ as a lower bound on the length of the shortest path to $\ell_i$ to prove an upper bound on the competitive ratio.

▶ Lemma 5. The target line $\ell_i$ intersects $Q_k$ and does not intersect $R_{k-3}$.

Proof. If $\ell_i$ intersects $Q_{k-1}$, then $\ell_i$ would have been found in phase $k-1$. Since $R_{k-3} \subset Q_{k-1}$, the lemma follows.

As $\ell_i$ does not intersect $R_{k-3}$ the closest point of $\ell_i$ to $O$ must be outside of $R_{k-3}$. But then the shortest path to $\ell_i$ must have length larger than $D_{k-3}$. Assume for contradiction that the closest point $p_i$ on $\ell_i$ has distance less than $D_{k-3}$. As in iteration $k-3$ we followed the paths that maximize (minimize) the $x$- and $y$-coordinate, $p_i$ could be reached and must thus be contained in $R_{k-3}$. Contradiction. Thus $D_{k-3}$ is a lower bound on the distance from $O$ to $\ell_i$, and $D_{k-3} = D_k/8$.

For an upper bound, we consider the distance we have travelled. Except for the last iteration, we traversed four paths of length $D_i$ in two directions in each iteration. Thus in previous iterations we traversed $8 \sum_{i=1}^{k-1} D_i$. In the last iteration in the worst-case we discover $\ell_i$ while traversing the fourth path all the way to its end. Hence we traverse three paths of length $D_k$ twice, and the last path of length $D_k$ once. The total travel is thus at most:

$$8 \sum_{i=1}^{k-1} D_i + 7D_k$$
Using the summation $\sum_{i=0}^{k-1} z^i = \frac{z^k-1}{z-1}$ and $D_i = 2^i d$ we can rewrite this to $15 \cdot 2^k d - 16d < 15D_k$. We thus upper-bound the competitive ratio by 120.

A more careful analysis shows that Lemma 4 is true even if we do not double $D_i$ but enlarge by only a factor $\sqrt{3}$. Let $D_1 = \sqrt{3}d$ and $D_i = \sqrt{3}D_{i-1}$ for $i \geq 2$, so $D_i = \sqrt{3}^i \cdot d$, and suppose $S$ finds $\ell_i$ in iteration $k$. Then $D_{k-3} = \sqrt{3}^{k-3} d$ is a lower bound for reaching $\ell_i$.

With the described strategy $S$ travels at most

$$8 \sum_{i=1}^{k-1} \sqrt{3}^i d + 7 \sqrt{3}^k d < 8 \cdot \frac{\sqrt{3}^k d}{\sqrt{3} - 1} + 7 \sqrt{3}^k d$$

The competitive ratio becomes

$$\frac{8 \frac{\sqrt{3}^k d}{\sqrt{3} - 1} + 7 \sqrt{3}^k d}{\sqrt{3}^{k-1} d} = (\frac{8}{\sqrt{3} - 1} + 7)\sqrt{3}^3 < 94$$

Another improvement comes from organizing the four traversals in a phase conveniently so that we do not have to go back to $O$ at the end. In every even phase $i$ we start with going to $p_i^{+x}$, then we do $p_i^{+y}$ and $p_i^{-y}$ in any order, and end with going to $p_i^{-x}$. In every odd phase $j$ we go to $p_j^{-x}$ first and to $p_j^{+x}$ last. It is easy to see that we do not have to go back at the end of any phase, because we go out over the exact same stretch in the next phase anyway. Instead of traversing $8D_i$ in a phase $i$, we now traverse $(7 - 1/\sqrt{3}) \cdot D_i$. This also holds for the last phase $D_k$. With some basic calculation we obtain:

> **Theorem 6.** A 79-competitive search strategy exists to find an unknown target line in an arrangement of lines.

Alternatively, we may also triple $D_i$ because then $R_i \subset Q_{i+1}$; a lower constant factor than 3 will not ensure that $R_i \subset Q_{i+1}$ so that will not give improvements. The competitive ratio we get is worse, however, than when using $\sqrt{3}$ and $R_i \subset Q_{i+2}$.

We note that if we know the exact distance to the line, we can use some of the ideas just given. By the observations above, we can find the unknown line by going three times as far in each direction. For the last direction $S$ does not need to go back, so in total we will find the line with competitive ratio 21.

### 4 A $c$-competitive search strategy on an unknown arrangement

In this section we consider the situation where the searcher $S$ does not know the arrangement beforehand. In particular, we assume $S$ learns the slope and intercept of a line, only when $S$ reaches it. The question arises whether we can adapt our competitive strategy to still realize a constant competitive ratio. The exact same strategy cannot be used, because we can no longer determine the points $p^{+y}$ and $p^{-y}$ before we start walking.

First of all, this problem suffers from a technicality that has been observed in similar problems: as soon as we decide to walk any distance from the starting location in some direction on the starting line, the target line could have been arbitrarily much closer in the other direction [2]. So a constant competitive ratio cannot exist. This technicality is commonly circumvented by assuming that the target line is at least some known - possibly extremely small - distance away from the start. We will assume this as well.

Assume the starting location is at the origin $O$ and lies on a horizontal line $\ell_0$. We start by finding the closest intersection to $O$. If it is at distance $d$, then we let $D_1 = 2d$. Similar to the strategy for known arrangements in iteration $i$ we aim to find the leftmost, rightmost,
Thus the largest (absolute) slope of any line traversed to get to while ending in the vertical slab $[-D,D]$ for the full length of the path to height $h_2$ and a path on $L_1$ of length $2tD + 2D$ (red) reaching height $h_1$. We show $h_1 \geq h_2$.

lowest, and highest point we can reach with distance $D$. We, however, choose our movement as to also discover a suitable set of “nearby” lines to which we must necessarily restrict our movement as we do not know about the existence of other lines. We show that with this smaller set of lines we can still achieve the height that we could have reached with knowledge of all lines; however, we traverse a constant factor further to ensure this.

We start by walking left and right from $O$ over a distance $tD$ for some constant $t \geq 1$ to be specified later. In doing so, we discover a subset $L_1$ of the lines. Let $L_2 = L \setminus L_1$, see Figure 7. Let $h_2$ be the height we could achieve within distance $D$ if we had full knowledge of the arrangement. Let the sequence of lines used to reach $h_2$ be $\ell_0, \ell_1, \ell_2, \ldots, \ell_j$. We know that $\ell_j$ is the steepest line among these, by the proof of Lemma 3.

We want to reach the highest point in the vertical slab $[-D,D]$ using lines from $L_1$ only. Clearly within a distance $D$ we can get at most as high as $h_2$. Instead we allow a traversal of distance $2tD + 2D$ along the lines of $L_1$. Let $h_1$ be the maximum height we can achieve while ending in the vertical slab $[-D,D]$ and when travelling over distance at most $2tD + 2D$ along only lines of $L_1$.

\textbf{Lemma 7.} $h_2 \leq h_1$ if $t \geq 2$.

\textbf{Proof.} Assume for contradiction that $h_2 > h_1$. Let $\ell_0, \ell_1, \ldots, \ell_j$ be the lines on a path of length $D$ to height $h_2$ on $L = L_1 \cup L_2$. Either $\ell_j \in L_1$ or $\ell_j \in L_2$.

Assume first that $\ell_j \in L_1$. Specifically then there is a point $p$ we can reach along $\ell_j$ that lies in the slab $[-D,D]$ at height $h_2$. However, $\ell_j$ intersects $\ell_0$ at most $tD$ from the origin. Thus we can follow $\ell_0$ to the intersection with $\ell_j$, and then follow $\ell_j$ to height $h_2$. As $h_2 \leq D$ this takes at most $tD + (t+1)D$ horizontal movement and $D$ vertical movement (see Figure 8). The total distance traversed along lines from $L_1$ is upper bounded by $2tD + 2D$, therefore $h_1 \geq h_2$. Contradiction.

Next, assume that $\ell_j \in L_2$. The line $\ell_j$ must intersect the rectangle $[-D,D] \times [0,h_2]$ since the path of length $D$ reaching $h_2$ cannot leave this rectangle. The maximum slope of a line $\ell_j \in L_2$ that intersects this rectangle is $\frac{h_2}{(t-1)D}$ as such a line must intersect $\ell_0$ at least $tD$ from the origin.

We must have that $\ell_j$ has the steepest absolute slope. If a previously traversed line had a steeper absolute slope we could follow it to get higher while staying in the slab $[-D,D]$. Thus the largest (absolute) slope of any line traversed to get to $h_2$ is $\frac{h_2}{(t-1)D}$. Take $t \geq 2$, then the largest slope is at most $\frac{h_2}{D}$. In the (unachievable) best case we traverse this slope for the full length of the path to height $h_2$, however then we still reach a height less than $h_2$. Contradiction.

Our constant competitive strategy, using $t = 2$, is therefore as follows: Go left over $2D$, then right over $4D$, then back to the starting point over $2D$, and form the set $L_1$. 

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure7.png}
\caption{The line sets $L_1$ and $L_2$, only some lines in $L_2$ are shown. Two paths maximizing the achieved height in the vertical slab $[-D,D]$: A path on $L_1 \cup L_2$ of length $D$ (blue) reaching height $h_2$ and a path on $L_1$ of length $2tD + 2D$ (red) reaching height $h_1$. We show $h_1 \geq h_2$.}
\end{figure}
Figure 8 Assume for contradiction that \( h_2 > h_1 \). The last line traversed to get to height \( h_2 \) within distance \( D \) on \( L_1 \cup L_2 \) must then be from \( L_2 \). If \( \ell_j \in L_1 \) then \( h_1 \geq h_2 \) as we can traverse only \( \ell_0 \) and \( \ell_j \) to reach the same height within distance \( 2tD + 2D \).

Use these lines, using distance \( 6D \) to get as high as possible in the vertical slab \([-D, D]\), and the same distance to get as low as possible, and back. In total we traverse a distance \( 8D + 12D + 12D = 32D \) in one phase. Then double \( D \) and repeat.

We once again argue that the true minimum and maximum \( x \) and \( y \) coordinates reachable in some phase \( i \) are covered completely by a quadrilateral on the discovered minima and maxima in a later phase. Let \( U_k \) be the quadrilateral created by our exploration of four paths on \( L_1 \) in phase \( k \).

Lemma 8. \( R_k \subset U_{k+2} \)

Proof. The proof of the lemma is identical to the proof of Lemma 4, with the following minor changes. See Figure 9 for an illustration of the proof.

Let \( r^{+y}_i \) be the highest point reachable in the slab \([-D_i, D_i]\) during phase \( i \). Once again let \( p^{+y}_i \) be the highest point achievable in distance \( D_i \) on the complete arrangement. From Lemma 7 we conclude that the \( y \)-coordinate of \( p^{+y}_i \) is less or equal than that of \( r^{+y}_i \). We also know that the \( x \)-coordinate of \( r^{+y}_{k+2} \) lies in the slab \([-D_{k+2}, D_{k+2}]\) so we do not need the triangle inequality of the proof. The proof follows directly.

We can now use the same method of analysis as for the case of a fully known line arrangement, except that we have to take into account that the searcher must move more in...
every phase. Once again we can scale the distance walked in an iteration by a factor of $\sqrt{3}$ instead of 2 to improve the bound. For a line found in iteration $i$ we traverse at most:

$$32 \sum_{i=1}^{k-1} D_i + 36D_k < 32\sqrt{3} \frac{d}{\sqrt{3} - 1} + 36\sqrt{3}d$$

A line found in iteration $i$ is at least at a distance of $D_k = \sqrt{3} d$. Thus we obtain the following result.

**Theorem 9.** A 414.2-competitive search strategy exists to find an unknown target line in an unknown arrangement of lines, where a line becomes known once we reach it.

## 5 Conclusions

We have developed and analyzed search strategies for reaching an unknown target line in an arrangements of lines. We did so by considering the competitive ratio: the worst-case ratio between the distance travelled by the searcher and the length of the shortest path from the searcher’s start location to the target line. We gave a search strategy for the case of known arrangements that achieves a competitive ratio of 79. Then we generalized our strategy so that it is competitive on line arrangements that are not known beforehand. The parameters of a line become known only when the line is reached. In this case we gave a 414.2-competitive search strategy. There is a considerable gap between the known lower bounds and upper bounds.

**Future work.** In our work we assumed that the speed on every line is the same. When we drop this assumption we do not know whether searching for a line can be done competitively even if we know all lines and all speeds. Certain properties still hold, for example, if we search for the largest $y$-coordinate, then we can get twice as far if we double the travel time. However, a diagonal with high speed may cause the furthest reachable point in both horizontal and vertical direction to be along this diagonal, essentially preventing growth of the explored region into other directions. When we search with a cost $T$ from $O$, the relevant points to visit seem to be the vertices of a convex polygon that is the convex hull of all points reachable at cost $T$. This polygon can have more than constantly many vertices so we cannot visit all in a phase. It is unclear how to choose a constant-size subset so that the resulting, smaller convex hull at least contains the full convex hull from a previous iteration.

We note that searching (connected) arrangements of simple geometric objects like line segments, circles, and half-lines cannot be done with a constant competitive strategy. But it is possible that if we impose restrictions on the arrangement, constant-competitive search strategies can be developed.

## References


Stabbing Pairwise Intersecting Disks by Five Points

Sariel Har-Peled
Department of Computer Science, University of Illinois, Urbana, IL 61801, USA
sariel@illinois.edu
https://orcid.org/0000-0003-2638-9635

Haim Kaplan
School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel
haimk@tau.ac.il

Wolfgang Mulzer
Institut für Informatik, Freie Universität Berlin, 14195 Berlin, Germany
mulzer@inf.fu-berlin.de
https://orcid.org/0000-0002-1948-5840

Liam Roditty
Department of Computer Science, Bar Ilan University, Ramat Gan 5290002, Israel
liamr@macs.biu.ac.il

Paul Seiferth
Institut für Informatik, Freie Universität Berlin, 14195 Berlin, Germany
pseiferth@inf.fu-berlin.de

Micha Sharir
School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel
michas@tau.ac.il

Max Willert
Institut für Informatik, Freie Universität Berlin, 14195 Berlin, Germany
willerma@inf.fu-berlin.de

Abstract

Suppose we are given a set $D$ of $n$ pairwise intersecting disks in the plane. A planar point set $P$ stabs $D$ if and only if each disk in $D$ contains at least one point from $P$. We present a deterministic algorithm that takes $O(n)$ time to find five points that stab $D$. Furthermore, we give a simple example of 13 pairwise intersecting disks that cannot be stabbed by three points. This provides a simple – albeit slightly weaker – algorithmic version of a classical result by Danzer that such a set $D$ can always be stabbed by four points.

2012 ACM Subject Classification Mathematics of computing → Combinatorics

Keywords and phrases Disk graph, piercing set, LP-type problem

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.50
Stabbing Pairwise Intersecting Disks by Five Points


Funding  Work on this paper was supported in part by grant 1367/2016 from the German-Israeli Science Foundation (GIF).

Acknowledgements  We would like to thank Timothy Chan for pointing us to the direct LP-type algorithm described in Lemma 4.7.

1 Introduction

Let $D$ be a set of $n$ disks in the plane. If every three disks in $D$ intersect, then Helly’s theorem shows that the whole intersection $\bigcap D$ of $D$ is nonempty [9, 10, 11]. In other words, there is a single point $p$ that lies in all disks of $D$, i.e., $p$ stabs $D$. More generally, when we know only that every pair of disks in $D$ intersect, there must be a point set $P$ of constant size such that each disk in $D$ contains at least one point in $P$. It is fairly easy to give an upper bound on the size of $P$, but for some time, the exact bound remained elusive. Eventually, in July 1956 at an Oberwolfach seminar, Danzer presented the answer: four points are always sufficient and sometimes necessary to stab any finite set of pairwise intersecting disks in the plane (see [5]). Danzer was not satisfied with his original argument, so he never formally published it. In 1986, he presented a new proof [5]. Previously, in 1981, Stachó had already given an alternative proof [15], building on a previous construction of five stabbing points [14]. This line of work was motivated by a result of Hadwiger and Debrunner, who showed that three points suffice to stab any finite set of pairwise intersecting unit disks [8]. In later work, these results were significantly generalized and extended, culminating in the celebrated $(p, q)$-theorem that was proven by Alon and Kleitman in 1992 [1]. See also a recent paper by Dumitrescu and Jiang that studies generalizations of the stabbing problem for translates and homothets of a convex body [6].

Danzer’s published proof [5] is fairly involved and uses a compactness argument, and part of it is based on an undetailed verification by computer. There seems to be no obvious way to turn it into an efficient algorithm for finding a stabbing set of size four. The two constructions of Stachó [15, 14] are simpler, but they start with three disks in $D$ with empty intersection and maximum inscribed circle. It is not clear to us how to find such a triple quickly (in, say, near-linear time). Here, we present a new argument that yields five stabbing points. Our proof is constructive, and it lets us find the stabbing set in deterministic linear time.

As for lower bounds, Grünbaum gave an example of 21 pairwise intersecting disks that cannot be stabbed by three points [7]. Later, Danzer reduced the number of disks to ten [5]. This example is close to optimal, because every set of eight disks can be stabbed by three points [14]. It is hard to verify Danzer’s lower bound example – even with dynamic geometry software, the positions of the disks cannot be visualized easily. Here, we present a simple construction that needs 13 disks and can be verified by inspection.

2 The Geometry of Pairwise Intersecting Disks

Let $D$ be a set of $n$ pairwise intersecting disks in the plane. A disk $D_i$ is given by its center $c_i$ and its radius $r_i$. To simplify the analysis, we make the following assumptions: (i) the radii of the disks are pairwise distinct; (ii) the intersection of any two disks has a nonempty interior; and (iii) the intersection of any three disks is either empty or has a nonempty interior. A simple perturbation argument can then handle the degenerate cases.
The lens of two disks \( D_i, D_j \in \mathcal{D} \) is the set \( L_{i,j} = D_i \cap D_j \). Let \( u \) be any of the two intersection points of \( \partial D_i \) and \( \partial D_j \). The angle \( \angle c_1 uc_j \) is called the lens angle of \( D_i \) and \( D_j \). It is at most \( \pi \). A finite set \( \mathcal{C} \) of disks is Helly if their common intersection \( \bigcap \mathcal{C} \) is nonempty. Otherwise, \( \mathcal{C} \) is non-Helly. We present some useful geometric lemmas.

\[ \text{Lemma 2.1.} \text{ Let } \{D_1, D_2, D_3\} \text{ be a set of three pairwise intersecting disks that is non-Helly. Then, the set contains two disks with lens angle larger than } 2\pi/3. \]

\[ \text{Proof.} \text{ Since } \{D_1, D_2, D_3\} \text{ is non-Helly, the lenses } L_{1,2}, L_{1,3}, \text{ and } L_{2,3} \text{ are pairwise disjoint. Let } u \text{ be the vertex of } L_{1,2} \text{ nearer to } D_3, \text{ and let } v, w \text{ be the analogous vertices of } L_{1,3}, \text{ and } L_{2,3} \text{ (see Figure 1, left). Consider the simple hexagon } c_1 wc_2 wc_3 v, \text{ and write } \angle u, \angle v, \text{ and } \angle w \text{ for its interior angles at } u, v, \text{ and } w. \text{ The sum of all interior angles is } 4\pi. \text{ Thus, } \angle u + \angle v + \angle w < 4\pi, \text{ so at least one angle is less than } 4\pi/3. \text{ It follows that the corresponding exterior angle at } u, v, \text{ or } w \text{ must be larger than } 2\pi/3. \]

\[ \text{Lemma 2.2.} \text{ Let } D_1 \text{ and } D_2 \text{ be two intersecting disks with } r_1 \geq r_2 \text{ and lens angle at least } 2\pi/3. \text{ Let } E \text{ be the unique disk with radius } r_1 \text{ and center } c, \text{ such that (i) the centers } c_1, c_2, \text{ and } c \text{ are collinear and } c \text{ lies on the same side of } c_1 \text{ as } c_2; \text{ and (ii) the lens angle of } D_1 \text{ and } E \text{ is exactly } 2\pi/3 \text{ (see Figure 1, right). Then, if } c_2 \text{ lies between } c_1 \text{ and } c, \text{ we have } D_2 \subseteq E. \]

\[ \text{Proof.} \text{ Let } x \in D_2. \text{ Since } c_2 \text{ lies between } c_1 \text{ and } c, \text{ the triangle inequality gives} \]

\[ |xc| \leq |xc_2| + |c_2c| = |xc_2| + |c_1c| - |c_1c_2|. \]  

(1)

Since \( x \in D_2 \), we get \(|xc_2| \leq r_2\). Also, since \( D_1 \) and \( E \) have radius \( r_1 \) each and lens angle \( 2\pi/3 \), it follows that \(|c_1c| = \sqrt{3}r_1\). Finally, \(|c_1c_2| = \sqrt{r_1^2 + r_2^2 - 2r_1r_2 \cos \alpha} \), by the law of cosines, where \( \alpha \) is the lens angle of \( D_1 \) and \( D_2 \). As \( \alpha \geq 2\pi/3 \) and \( r_1 \geq r_2 \), we get \( \cos \alpha \leq -1/2 = (\sqrt{3} - 3/2) - \sqrt{3} + 1 \leq (\sqrt{3} - 3/2)r_1/r_2 - \sqrt{3} + 1 \). As such, we have

\[ |c_1c_2|^2 = r_1^2 + r_2^2 - 2r_1r_2 \cos \alpha \geq r_1^2 + r_2^2 - 2r_1r_2 \left( (\sqrt{3} - 3/2)\frac{r_1}{r_2} - \sqrt{3} + 1 \right) \]

\[ = r_1^2 - 2(\sqrt{3} - 3/2)r_1^2 + 2(-\sqrt{3} + 1)r_1r_2 + r_2^2 \]

\[ = (1 - 2\sqrt{3} + 3)r_1^2 + 2(-\sqrt{3} + 1)r_1r_2 + r_2^2 = (r_1(\sqrt{3} - 1) + r_2)^2. \]

Plugging this into Eq. (1) gives \(|xc| \leq r_2 + \sqrt{3}r_1 - (r_1(\sqrt{3} - 1) + r_2) = r_1\), i.e., \( x \in E \).

\[ \text{Lemma 2.3.} \text{ Let } D_1 \text{ and } D_2 \text{ be two intersecting disks with equal radius } r \text{ and lens angle } 2\pi/3. \text{ There is a set } P \text{ of four points so that any disk } F \text{ of radius at least } r \text{ that intersects both } D_1 \text{ and } D_2 \text{ contains a point of } P. \]
Proof. Consider the two tangent lines of \( D_1 \) and \( D_2 \), and let \( p \) and \( q \) be the midpoints on these lines between the respective two tangency points. We set \( P = \{ c_1, c_2, p, q \} \) (see Figure 2, left).

Given the disk \( F \) that intersects both \( D_1 \) and \( D_2 \), we shrink its radius, keeping its center fixed, until either the radius becomes \( r \) or until \( F \) is tangent to \( D_1 \) or \( D_2 \). Suppose the latter case holds and \( F \) is tangent to \( D_1 \). We move the center of \( F \) continuously along the line spanned by the center of \( F \) and \( c_1 \) towards \( c_1 \), decreasing the radius of \( F \) to maintain the tangency. We stop when either the radius of \( F \) reaches \( r \) or \( F \) becomes tangent to \( D_2 \). We obtain a disk \( G \subseteq F \) with center \( c = (c_x, c_y) \) so that either: (i) \( \text{radius}(G) = r \) and \( G \) intersects both \( D_1 \) and \( D_2 \); or (ii) \( \text{radius}(G) \geq r \) and \( G \) is tangent to both \( D_1 \) and \( D_2 \). Since \( G \subseteq F \), it suffices to show that \( G \cap P \neq \emptyset \). We introduce a coordinate system, setting the origin \( o \) midway between \( c_1 \) and \( c_2 \), so that the y-axis passes through \( p \) and \( q \). Then, as in Figure 2 (left), we have \( c_1 = (-\sqrt{3}r/2, 0), c_2 = (\sqrt{3}r/2, 0), q = (0, r), \) and \( p = (0, -r) \).

For case (i), let \( D_1^2 \) be the disk of radius \( 2r \) centered at \( c_1 \), and \( D_2^2 \) the disk of radius \( 2r \) centered at \( c_2 \). Since \( G \) has radius \( r \) and intersects both \( D_1 \) and \( D_2 \), its center \( c \) has distance at most \( 2r \) from both \( c_1 \) and \( c_2 \), i.e., \( c \in D_1^2 \cap D_2^2 \). Let \( D_p \) and \( D_q \) be the two disks of radius \( r \) centered at \( p \) and \( q \). We will show that \( D_1^2 \cap D_2^2 \subseteq D_1^p \cup D_2^p \cup D_p \cup D_q \). Then it is immediate that \( G \cap P \neq \emptyset \). By symmetry, it is enough to focus on the upper-right quadrant \( Q = \{(x, y) \mid x \geq 0, y \geq 0 \} \). We show that all points in \( D_1^2 \cap Q \) are covered by \( D_2^2 \cup D_q \).

Without loss of generality, we assume that \( r = 1 \). Then, the two intersection points of \( D_1^2 \) and \( D_q \) are:
\[
\begin{align*}
\begin{array}{l}
\text{for } D_1^2 \\
\text{and } D_q
\end{array}
\end{align*}
\]
\[
\begin{align*}
\begin{array}{l}
(5\sqrt{3} - 2\sqrt{28}, 38 + 3\sqrt{28}) \approx (-0.36, 1.93) \\
(\sqrt{3} + 2\sqrt{28}, 38 - 3\sqrt{28}) \approx (0.98, 0.78)
\end{array}
\end{align*}
\]

and the two intersection points of \( D_2^2 \) and \( D_2 \) are:
\[
\begin{align*}
\begin{array}{l}
\text{for } D_2^2 \\
\text{and } D_2
\end{array}
\end{align*}
\]
\[
\begin{align*}
\begin{array}{l}
s_1 = (\frac{\sqrt{3} - 2\sqrt{28}}{2}, 1) \approx (0.87, 1) \\
s_2 = (\frac{\sqrt{3} + 2\sqrt{28}}{2}, -1) \approx (0.87, -1)
\end{array}
\end{align*}
\]

Let \( \gamma \) be the boundary curve of \( D_1^2 \) in \( Q \). Since \( r_1, s_2 \notin Q \) and since \( r_2 \in D_2 \) and \( s_1 \in D_q \), it follows that \( \gamma \) does not intersect the boundary of \( D_2 \cup D_q \) and hence \( \gamma \subseteq D_2 \cup D_q \). Furthermore, the subsegment of the y-axis from \( o \) to the start point of \( \gamma \) is contained in \( D_q \), and the subsegment of the x-axis from \( o \) to the endpoint of \( \gamma \) is contained in \( D_2 \). Hence, the boundary of \( D_1^2 \cap Q \) lies completely in \( D_2 \cup D_q \), and since \( D_2 \cup D_q \) is simply connected, it follows that \( D_1^2 \cap Q \subseteq D_2 \cup D_q \), as desired.

For case (ii), since \( G \) is tangent to \( D_1 \) and \( D_2 \), the center \( c \) of \( G \) is on the perpendicular bisector of \( c_1 \) and \( c_2 \), so the points \( p, a, q \) and \( c \) are collinear. Suppose without loss of generality that \( c < 0 \). Then, it is easily checked that \( c \) lies above \( q \), and radius(\( G \)) + \( r = |c_1 c| \geq |o c| = r + |q c| \), so \( q \in G \).

\begin{lemma}
Consider two intersecting disks \( D_1 \) and \( D_2 \) with \( r_1 \geq r_2 \) and lens angle at least \( 2\pi/3 \). Then, there is a set \( P \) of four points such that any disk \( F \) of radius at least \( r_1 \) that intersects both \( D_1 \) and \( D_2 \) contains a point of \( P \).
\end{lemma}
Proof. Let $\ell$ be the line through $c_1$ and $c_2$. Let $E$ be the disk of radius $r_1$ and center $c \in \ell$ that satisfies the conditions (i) and (ii) of Lemma 2.2. Let $P = \{c_1, c, p, q\}$ as in the proof of Lemma 2.3, with respect to $D_1$ and $E$ (see Figure 1, right). We claim that

$$D_1 \cap F \neq \emptyset \land D_2 \cap F \neq \emptyset \Rightarrow E \cap F \neq \emptyset. \quad (*)$$

Once (*) is established, we are done by Lemma 2.3. If $D_2 \subseteq E$, then (*) is immediate, so assume that $D_2 \nsubseteq E$. By Lemma 2.2, $c$ lies between $c_1$ and $c_2$. Let $k$ be the line through $c$ perpendicular to $\ell$, and let $k^+$ be the open halfplane bounded by $k$ with $c_1 \in k^+$ and $k^-$ the open halfplane bounded by $k$ with $c_1 \notin k^-$. Since $|c_1c| = \sqrt{3}r_1 > r_1$, we have $D_1 \subset k^+$ (see Figure 2, right). Recall that $F$ has radius at least $r_1$ and intersects $D_1$ and $D_2$. We distinguish two cases: (i) there is no intersection of $F$ and $D_2$ in $k^+$, and (ii) there is an intersection of $F$ and $D_2$ in $k^+$.

For case (i), let $x$ be any point in $D_1 \cap F$. Since we know that $D_1 \subset k^+$, we have $x \in k^+$. Moreover, let $y$ be any point in $D_2 \cap F$. By assumption (i), $y$ is not in $k^+$, but it must be in the infinite strip defined by the two tangents of $D_1$ and $E$. Thus, the line segment $\overline{xy}$ intersects the diameter segment $k \cap E$. Since $F$ is convex, the intersection of $\overline{xy}$ and $k \cap E$ is in $F$, so $E \cap F \neq \emptyset$.

For case (ii), fix $x \in D_2 \cap F \cap k^+$ arbitrarily. Consider the triangle $\Delta xcc_2$. Since $x \in k^+$, the angle at $c$ is at least $\pi/2$ (Figure 2, right). Thus, $|xc| \leq |x_2c_2|$. Also, since $x \in D_2$, we know that $|xc_2| \leq r_2 \leq r_1$. Hence, $|xc| \leq r_1$, so $x \in E$ and (*) follows, as $x \in E \cap F$. □

3 Existence of Five Stabbing Points

With the tools from Section 2, we can now show that there is a stabbing set with five points.

Theorem 3.1. Let $D$ be a set of $n$ pairwise intersecting disks in the plane. There is a set $P$ of five points such that each disk in $D$ contains at least one point from $P$.

Proof. If $D$ is Helly, there is a single point that lies in all disks of $D$. Thus, assume that $D$ is non-Helly, and let $D_1, D_2, \ldots, D_n$ be the disks in $D$ ordered by increasing radius. Let $i^*$ be the smallest index with $\bigcap_{i \leq i^*} D_i = \emptyset$. By Helly’s theorem [9, 10, 11], there are indices $j, k < i^*$ such that $\{D_{i'}, D_j, D_k\}$ is non-Helly. By Lemma 2.1, two disks in $\{D_{i'}, D_j, D_k\}$ have lens angle at least $2\pi/3$. Applying Lemma 2.4 to these two disks, we obtain a set $P'$ of four points so that every disk $D_i$ with $i \geq i^*$ contains at least one point from $P'$. Furthermore, by definition of $i^*$, we have $\bigcap_{i < i^*} D_i \neq \emptyset$, so there is a point $q$ that stab every disk $D_i$ with $i < i^*$. Thus, $P = P' \cup \{q\}$ is a set of five points that stabs every disk in $D$, as desired. □

4 Algorithmic Considerations

Theorem 3.1 leads to a simple $O(n \log n)$ time deterministic algorithm for finding a stabbing set of size 5: we sort the disks in $D$ by radius, and we insert the disks one by one, while maintaining their intersection. Once the intersection becomes empty, we can use the method from Theorem 3.1 to find the stabbing set (otherwise, $D$ is Helly, and we have a single stabbing point). As we will see next, there is also a deterministic linear time algorithm, using the LP-type framework by Sharir and Welzl [13, 3].

The LP-type framework. An LP-type problem $(\mathcal{H}, w, \leq)$ is an abstract generalization of a low-dimensional linear program. It consists of a finite set of constraints $\mathcal{H}$, a weight function $w : 2^\mathcal{H} \to W$, and a total order $(W, \leq)$ on the weights. The weight function $w$ assigns a weight to each subset of constraints. It must fulfill the following three axioms:
Geometric observations. The distance between two closed sets $A, B \subseteq \mathbb{R}^2$ is defined as $d(A, B) = \min\{d(a, b) \mid a \in A, b \in B\}$. From now on, we assume that all points in $\bigcup D$ have positive $y$-coordinates. This can be ensured with linear overhead by an appropriate translation of the input. We denote by $D_\infty$ the closed halfplane below the $x$-axis. It is

---

5 Here, we follow the presentation of Chazelle [3]. Sharir and Welzl [13] do not require property (i) of a violation test. Instead, they need an additional basis computation primitive: given a basis $B$ and a constraint $H \in H$, find a basis for $B \cup \{H\}$. Given a violation test with property (i), a basis computation primitive can easily be implemented by brute force enumeration.
interred as a disk with radius $\infty$ and center at $(0, -\infty)$. For $C \subseteq D$ we set $\overline{C} = C \cup \{D_\infty\}$. Observe that for $C_1 \subseteq C_2 \subseteq \overline{D}$, if $C_1$ is non-Helly, then $C_2$ is non-Helly. Furthermore, for $r \in \mathbb{R} \cup \{\infty\}$ and $C \subseteq \overline{D}$, we define $C_{\leq r}$ (resp., $C_{< r}$) as the set of all disks in $C$ with radius at most (resp., smaller than) $r$. Let $C \subseteq D$ be Helly. A disk $D \in \overline{D}$ is a destroyer of $C$ if $C \cup \{D\}$ is non-Helly. Observe that $D_\infty$ is a destroyer for every Helly subset of $D$. Now, let $C \subseteq D$ be an arbitrary subset of $D$ (either Helly or non-Helly). We say $D \in \overline{D}$ is the smallest destroyer of $C$ if $C_{\leq r}$ is Helly and $\overline{C}_{\leq r}$ is non-Helly, where $r$ is the radius of $D$. Note that $D$ is the unique largest disk in $\overline{C}_{\leq r}$. Furthermore, $D$ is the smallest disk in $\overline{C}$ that causes a non-Helly triple. If $C$ is Helly, then $D = D_\infty$. See Figure 3 for an example. We can make the following two observations.

Lemma 4.1. Let $C \subseteq D$ be Helly and $D \in \overline{D}$ a destroyer of $C$. Then, the point $v \in \bigcap C$ with minimum distance to $D$ is unique.

Proof. Suppose there are two distinct points $v \neq w \in \bigcap C$ with $d(v, D) = d(\bigcap C, D) = d(w, D)$. Since $\bigcap C$ is convex, the segment $\overline{vw}$ lies in $\bigcap C$. Now, if $D \neq D_\infty$, then every point in the relative interior of $\overline{vw}$ is strictly closer to $D$ than $v$ and $w$. If $D = D_\infty$, then all points in $\overline{vw}$ have the same distance to $D$, but since $\bigcap C$ is strictly convex, the relative interior of $\overline{vw}$ lies in the interior of $\bigcap C$, so there must be a point in $\bigcap C$ that is closer to $D$ than $v$ and $w$. In either case, we obtain a contradiction to the assumption $v \neq w$ and $d(v, D) = d(\bigcap C, D) = d(w, D)$. The claim follows.

The unique point $v \in \bigcap C$ with minimum distance to a destroyer $D$ is called the extreme point for $C$ and $D$ (see Figure 3).

Lemma 4.2. Let $C_1 \subseteq C_2 \subseteq D$ be two Helly sets and $D \in \overline{D}$ a destroyer of $C_1$ (and thus of $C_2$). Let $v \in \bigcap C_1$ be the extreme point for $C_1$ and $D$. We have $d(\bigcap C_1, D) \leq d(\bigcap C_2, D)$. In particular, if $v \in \bigcap C_2$, then $d(\bigcap C_1, D) = d(\bigcap C_2, D)$ and $v$ is also the extreme point for $C_2$. If $v \not\in \bigcap C_2$, then $d(\bigcap C_1, D) < d(\bigcap C_2, D)$.

Proof. The first claim holds trivially: let $w \in \bigcap C_2$ be the extreme point for $C_2$ and $D$. Since $C_1 \subseteq C_2$, it follows that $w \in \bigcap C_1$, so $d(\bigcap C_1, D) \leq d(w, D) = d(\bigcap C_2, D)$. If $v \in \bigcap C_2$, then $d(\bigcap C_1, D) \leq d(\bigcap C_2, D) \leq d(v, D) = d(\bigcap C_1, D)$, so $v = w$, by Lemma 4.1. If $v \not\in \bigcap C_2$, then $d(\bigcap C_1, D) < d(\bigcap C_2, D)$, by Lemma 4.1 and the fact that $C_1 \subseteq C_2$. See Figure 4.
Figure 5: Monotonicity: In both cases, \{\(D_1, D_2, D_3\)\} is non-Helly with smallest destroyer \(D_3\). Adding a disk \(E\) either decreases the radius of the smallest destroyer (left) or increases the distance to the smallest destroyer (right).

Figure 6: A basis can either be a non-Helly triple (left), a pair of intersecting disks \(E\) and \(F\) where the point of minimum \(y\)-coordinate in \(E \cap F\) is a vertex (middle), or a single disk.

Let \(\mathcal{C}\) be a subset of \(\mathcal{D}\). The radius of the smallest destroyer \(D\) of \(\overline{\mathcal{C}}\) is denoted by \(\text{rad}(\mathcal{C})\). Note that \(\text{rad}(\mathcal{C}) \in \mathbb{R}_{>0} \cup \{\infty\}\). Moreover, let \(\text{dist}(\mathcal{C})\) be the distance between \(D\) and the set \(\bigcap \mathcal{C}_{\leq \text{rad}(\mathcal{C})}\), i.e., \(\text{dist}(\mathcal{C}) = d(\bigcap \mathcal{C}_{\leq \text{rad}(\mathcal{C})}, D)\). Then, \(\mathcal{C}\) is Helly if and only if \(\text{rad}(\mathcal{C}) = \infty\). In this case, \(\text{dist}(\mathcal{C})\) is the distance between \(\bigcap \mathcal{C}\) and the \(x\)-axis. We define the weight \(w(\mathcal{C})\) of \(\mathcal{C}\) as \(w(\mathcal{C}) = (\text{rad}(\mathcal{C}), -\text{dist}(\mathcal{C}))\), and we denote by \(\leq\) the lexicographic order on \(\mathbb{R}^2\). Chan observed, in a slightly different context, that \((\mathcal{D}, w, \leq)\) is LP-type [2]. However, Chan’s paper does not contain a detailed proof for this fact. Thus, in the following lemmas, we show that the three LP-type axioms hold.

▶ Lemma 4.3. For any \(\mathcal{C} \subseteq \mathcal{D}\) and \(E \in \mathcal{D}\), we have \(w(\mathcal{C} \cup \{E\}) \leq w(\mathcal{C})\).

Proof. Set \(\mathcal{C}^* = \mathcal{C} \cup \{E\}\). Let \(D\) be the smallest destroyer of \(\overline{\mathcal{C}}\), and let \(r = \text{rad}(\mathcal{C})\) be the radius of \(D\). Then, \(D\) is the largest disk in \(\mathcal{C}_{\leq r}\). The set \(\mathcal{C}_{\leq r}\) is non-Helly. Adding \(E\) does not change this, i.e., \(\mathcal{C}_{\leq r}'\) is also non-Helly. Thus, the smallest destroyer of \(\mathcal{C}_{\leq r}'\) is either \(D\) or some smaller disk in \(\mathcal{C}_{\leq r}\). In the latter case, we have \(\text{rad}(\mathcal{C}^*) < \text{rad}(\mathcal{C})\). In the former case, we have \(\text{rad}(\mathcal{C}^*) = \text{rad}(\mathcal{C})\), and Lemma 4.2 gives \(-\text{dist}(\mathcal{C}^*) = -d(\bigcap \mathcal{C}_{\leq r}, D) \leq -d(\bigcap \mathcal{C}_{\leq r}, D) = -\text{dist}(\mathcal{C})\). In either case, \(w(\mathcal{C}^*) \leq w(\mathcal{C})\). See Figure 5 for an illustration.

▶ Lemma 4.4. For any \(\mathcal{C} \subseteq \mathcal{D}\), there is a set \(\mathcal{B} \subseteq \mathcal{C}\) with \(|\mathcal{B}| \leq 3\) and \(w(\mathcal{B}) = w(\mathcal{C})\).
Proof. Let $D$ be the smallest destroyer of $\overline{C}$. Let $r = \text{rad}(C)$ be the radius of $D$, and let $v \in \bigcap C_{\text{cr}}$ be the extreme point for $C_{\text{cr}}$ and $D$. By general position, there are at most two disks $E, F \in C_{\text{cr}}$ with $v \in \partial (E \cap F)$. Note that $E$ and $F$ may be the same disk.

Set $B = \{D, E, F\} \setminus \{D_\infty\}$. There are three possibilities. If $C$ is non-Helly, then $D \neq D_\infty$ and $B$ is a non-Helly triple (indeed, as the disks in $D$ are pairwise intersecting, the extreme point $v$ must lie at the intersection of two disk boundaries). If $C$ is Helly, then $D = D_\infty$ and $|B| = 2$. If $|B| = 2$, then $v$ is the vertex of $\partial (E \cap F)$ with minimum $y$-coordinate. If $|B| = 1$, then $v$ is the point on $\partial E$ with minimum $y$-coordinate. In either case, $\text{dist}(B)$ is the value of the smallest $y$-coordinate of a point in $\bigcap B$. See Figure 6 for an illustration.

We claim that $w(B) = w(C)$. Firstly, $\text{rad}(B) = \text{rad}(C)$, because $B$ and $C$ have the same smallest destroyer. Secondly, we show $\text{dist}(B) = \text{dist}(C)$: since $B_{\text{cr}} \subseteq C_{\text{cr}}$, by Lemma 4.2, we get $\text{dist}(B) = d(\bigcap B_{\text{cr}}, D) \leq d(\bigcap C_{\text{cr}}, D) = \text{dist}(C)$. Suppose that $\text{dist}(B) < \text{dist}(C)$. Then, there is a point $w \in E \cap F$ with $d(w, D) < d(v, D)$. Furthermore, by general position and since $v$ is the intersection of two disk boundaries, there is a relatively open neighborhood $N$ around $v$ in $\bigcap C_{\text{cr}}$, such that $N$ is also relatively open in $E \cap F$. Since $E \cap F$ is convex, there is a point $x \in N$ that also lies in the relative interior of the line segment $\overline{vw}$. Then, $d(x, D) < d(v, D)$ and $x \in \bigcap C_{\text{cr}}$, which is impossible, as $v$ is the extreme point.

The set $B$ is actually a basis for $C$: if $B$ is a non-Helly triple, then removing any disk from $B$ creates a Helly set and increases the radius of the smallest destroyer to $\infty$. If $|B| = 2$, then $D_\infty$ is the smallest destroyer of $B$ and the minimality follows directly from the definition.

Lemma 4.5. Let $B \subseteq C \subseteq D$ with $w(B) = w(C)$ and let $E \in D$. Then, if $w(B \cup \{E\}) = w(B)$ we also have $w(C \cup \{E\}) = w(C)$.

Proof. Set $C^* = C \cup \{E\}, B^* = B \cup \{E\}$. Let $r = \text{rad}(C)$ and $D$ the smallest destroyer of $\overline{C}$. Since $w(C) = w(B) = w(B^*)$, we have that $D$ is also the smallest destroyer of $B$ and of $B^*$. If $E$ has radius $r$, then $E$ cannot be the smallest destroyer of $\overline{C}$, so $w(C^*) = w(C)$. Assume that $E$ has radius $< r$. Let $v$ be the extreme point of $C_{\text{cr}}$ and $D$. Since $w(B^*) = w(B)$, we know that $d(\bigcap B_{\text{cr}}, D) = d(\bigcap B^*_{\text{cr}}, D) = d(v, D)$. Now, Lemma 4.2 implies $v \in E$, since $E \in B^*_{\text{cr}}$. Thus, the set $C^*_{\text{cr}} = C_{\text{cr}} \cup \{E\}$ is Helly. Furthermore, $\overline{C^*_{\text{cr}}}$ is non-Helly, because the subset $\overline{C^*_{\text{cr}}}$ is non-Helly. Therefore, $D$ is also the smallest destroyer of $\overline{C^*}$, so $\text{rad}(C^*) = r = \text{rad}(C)$. Finally, since $B^*_{\text{cr}} \subseteq C^*_{\text{cr}}$, we can use Lemma 4.2 to derive

$$d\left(\bigcap C_{\text{cr}}, D\right) = d\left(\bigcap B^*_{\text{cr}}, D\right) \leq d\left(\bigcap C^*_{\text{cr}}, D\right) = d(v, D) = d\left(\bigcap C_{\text{cr}}, D\right).$$

Next, we describe a violation test for $(D, w, \leq)$: given a set $B \subseteq D$ and a disk $E \in D$ with radius $r$, determine (i) whether $B$ is a basis for some subset of $D$, and (ii) whether $E$ violates $B$, i.e., whether $w(B \cup \{E\}) < w(B)$. This is done as follows:

- If (i) $|B| > 3$; or (ii) $|B| = 3$ and $B$ is Helly; or (iii) $|B| = 2$ and the $y$-minimum of $\bigcap B$ is also the $y$-minimum of a single disk of $B$, return “$B$ is not a basis”.
- If $|B| = 1$, then, if the $y$-minimum in $E \cap \bigcap B$ differs from the $y$-minimum in $\bigcap B$, return “$E$ violates $B^*$”; otherwise, return “$E$ does not violate $B^*$”.
- If $B = \{D_1, D_2\}$, find the $y$-minimum $v$ of $D_1 \cap D_2$ and return “$E$ violates $B^*$” if $v \notin E$, and “$E$ does not violate $B^*$”, otherwise.
- Finally, if $B = \{D, D_1, D_2\}$ is non-Helly with smallest destroyer $D$,¹ let $r = \text{rad}(B)$ be the radius of $D$ and $r'$ be the radius of $E$.

¹ Note that since $B$ is a subset of $D$ and since $B$ is non-Helly, the smallest destroyer $D$ of $B$ cannot be the disk $D_\infty$.  

ISAAC 2018
If \( r' > r \), return “\textit{E does not violate B}”.

If \( r' < r \), find the vertex \( v \) of \( D_1 \cap D_2 \) that minimizes the distance to \( E \) and return “\textit{E violates B}” if \( v \notin E \), and “\textit{E does not violate B}”, otherwise.

The violation test obviously needs constant time. Finally, to apply the algorithm of Chazelle and Matoušek, we still need to check that the range space \((D, R)\) with \( R = \{ \text{vio}(B) \mid B \text{ is a basis of a subset in } D \} \) has bounded VC dimension.

\textbf{Lemma 4.6.} The range space \((D, R)\) has VC-dimension at most 3.

\textbf{Proof.} The discussion above shows that for any basis \( B \), there is a point \( v_B \in \mathbb{R}^2 \) such that \( E \in D \) violates \( B \) if and only if \( v_B \notin E \). Thus, for any \( v \in \mathbb{R}^2 \), let \( R' = \{ D \in D \mid v \notin D \} \) and let \( R'' = \{ v \in \mathbb{R}^2 \mid v \in R' \} \). Since \( R \subseteq R'' \), it suffices to show that \((D, R'')\) has bounded VC-dimension. For this, consider the complement range space \((D, R''')\) with \( R''' = \{ v \in \mathbb{R}^2 \mid v \notin R'' \} \) and \( \{D \in D \mid v \in D\} \), for \( v \in \mathbb{R}^2 \). It is well known that \((D, R'')\) and \((D, R''')\) have the same VC-dimension [3], and that \((D, R''')\) has VC-dimension 3 (e.g., this follows from the classic homework exercise that there is no planar Venn-diagram for four sets).

Finally, the following lemma summarizes discussion so far.

\textbf{Lemma 4.7.} Given a set \( D \) of \( n \) pairwise intersecting disks in the plane, we can decide in \( O(n) \) deterministic time whether \( D \) is Helly. If so, we can compute a point in \( \bigcap D \) in \( O(n) \) deterministic time. If not, we can compute the smallest destroyer \( D \) of \( D \) and two disks \( E, F \in D_{<r} \), that form a non-Helly triple with \( D \). Here, \( r \) is the radius of \( D \).

\textbf{Proof.} Since \((D, w, \leq)\) is LP-type, the violation test needs \( O(1) \) time, and the VC-dimension of \((D, R)\) is bounded, we can apply the deterministic algorithm of Chazelle and Matoušek [4] to compute \( w(D) = (\text{rad}(D), \text{dist}(D)) \) and a corresponding basis \( B \) in \( O(n) \) time. Then, \( D \) is Helly if and only if \( \text{rad}(D) = \infty \). If \( D \) is Helly, then \(|B| \leq 2 \). We compute the unique point \( v \in \bigcap B \) with \( d(v, D_{\infty}) = d(\bigcap B, D_{\infty}) \). Since \( B \subseteq D \) and \( d(\bigcap B, D_{\infty}) = d(\bigcap D, D_{\infty}) \), we have \( v \in \bigcap D \) by Lemma 4.2. We output \( v \). If \( D \) is non-Helly, we simply output \( B \), because \( B \) is a non-Helly triple with the smallest destroyer \( D \) of \( D \) and two disks \( E, F \in D_{<r} \), where \( r \) is the radius of \( D \).

\textbf{Theorem 4.8.} Given a set \( D \) of \( n \) pairwise intersecting disks in the plane, we can find in \( O(n) \) time a set \( P \) of five points such that every disk of \( D \) contains at least one point of \( P \).

\textbf{Proof.} Using the algorithm from Lemma 4.7, we decide whether \( D \) is Helly. If so, we return the point computed by the algorithm. Otherwise, the algorithm gives us a non-Helly triple \( \{D, E, F\} \), where \( D \) is the smallest destroyer of \( D \) and \( E, F \in D_{<r} \), with \( r \) being the radius of \( D \). Since \( D_{<r} \) is Helly, we can obtain in \( O(n) \) time a stabbing point \( q \in \bigcap D_{<r} \) by using the algorithm from Lemma 4.7 again. Next, by Lemma 2.1, there are two disks in \( \{D, E, F\} \) whose lens angle is at least \( 2\pi/3 \). Let \( P' \) be the set of four points from the proof of Lemma 2.4. Then, \( P = P' \cup \{q\} \) is a set of five points that stabs every disk in \( D \).

5 A Simple Lower Bound

We now exhibit a set of 13 pairwise intersecting disks in the plane such that no point set of size three can pierce all of them. The construction begins with an inner disk \( A \) of radius 1 and three larger disks \( D_1, D_2, D_3 \) of equal radius, so that \( A \) is tangent to all three disks and so that each two disks are tangent to each other. For \( i = 1, 2, 3 \), we denote the contact point of \( A \) and \( D_i \) by \( \xi_i \).
We add six more disks as follows. For $i = 1, 2, 3$, we draw the two common outer tangents to $A$ and $D_i$, and denote by $T_i^-$ and $T_i^+$ the halfplanes that are bounded by these tangents and are openly disjoint from $A$. The labels $T_i^-$ and $T_i^+$ are chosen such that the points of tangency between $A$ and $T_i^-$, $D_i$, and $T_i^+$, appear along $\partial A$ in this counterclockwise order.

One can show that the nine points of tangency between $A$ and the other disks and tangents are pairwise distinct (see Figure 7). We regard the six halfplanes $T_i^-$, $T_i^+$, for $i = 1, 2, 3$, as (very large) disks; in the end, we can apply a suitable inversion to turn the disks and halfplanes into actual disks, if so desired.

Finally, we construct three additional disks $A_1$, $A_2$, $A_3$. To construct $A_i$, we slightly expand $A$ into a disk $A_i'$ of radius $1 + \varepsilon_1$, while keeping the tangency with $D_i$ at $\xi_i$. We then roll $A_i'$ clockwise along $D_i$, by a tiny angle $\varepsilon_2 \ll \varepsilon_1$, to obtain $A_i$.

This gives a set of 13 disks. For sufficiently small $\varepsilon_1$ and $\varepsilon_2$, we can ensure the following properties for each $A_i$: (i) $A_i$ intersects all other 12 disks; (ii) the nine intersection regions $A_i \cap D_j$, $A_i \cap T_j^-$, $A_i \cap T_j^+$, for $j = 1, 2, 3$, are pairwise disjoint; and (iii) $\xi_i \notin A_i$.

**Theorem 5.1.** The construction yields a set of 13 disks that cannot be stabbed by 3 points.

**Proof.** Consider any set $P$ of three points. Set $A^* = A \cup A_1 \cup A_2 \cup A_3$. If $P \cap A^* = \emptyset$, we have unstabbed disks, so suppose that $P \cap A^* \neq \emptyset$. For $p \in P \cap A^*$, property (ii) implies that $p$ stabs at most one of the nine remaining disks $D_j$, $T_j^+$ and $T_j^-$, for $j = 1, 2, 3$. Thus, if $P \subset A^*$, we would have unstabbed disks, so we may assume that $|P \cap A^*| \in \{1, 2\}$.

Suppose first that $|P \cap A^*| = 2$. As just argued, at most two of the remaining disks are stabbed by $P \cap A^*$. The following cases can then arise.

(a) None of $D_1$, $D_2$, $D_3$ is stabbed by $P \cap A^*$. Since $\{D_1, D_2, D_3\}$ is non-Helly and a non-Helly set must be stabbed by at least two points, at least one disk remains unstabbed.

(b) Two disks among $D_1$, $D_2$, $D_3$ are stabbed by $P \cap A^*$. Then the six unstabbed halfplanes form many non-Helly triples, e.g., $T_1^-$, $T_2^+$, and $T_3^-$, and again, a disk remains unstabbed.

(c) The set $P \cap A^*$ stabs one disk in $\{D_1, D_2, D_3\}$ and one halfplane. Then, there is (at least) one disk $D_i$ such that $D_i$ and its two tangent halfplanes $T_i^-$, $T_i^+$ are all unstabbed by $P \cap A^*$. Then, $\{D_i, T_i^-, T_i^+\}$ is non-Helly, and at least two more points are needed to stab it.

Suppose now that $|P \cap A^*| = 1$, and let $P \cap A^* = \{p\}$. We may assume that $p$ stabs all three disks $A_1$, $A_2$, $A_3$, since otherwise a disk would stay unstabbed. At most one of the nine remaining disks is stabbed by $p$. Thus, $p \notin \{\xi_1, \xi_2, \xi_3\}$, so the other disk that it stabs (if any) must be a halfplane. That is, $p$ does not stab any of $D_1$, $D_2$, $D_3$. Since $\{D_1, D_2, D_3\}$ is non-Helly, it requires two stabbing points. Moreover, since $|P \setminus \{p\}| = 2$, we may assume that
one point \( q \) of \( P \setminus A^* \) is the point of tangency of two of these disks, say \( q = D_2 \cap D_3 \). Then, \( q \) stabs only two of the six halfplanes, say, \( T_{1}^- \) and \( T_{2}^+ \). But then, \( \{D_1, T_2^+, T_3^-\} \) is non-Helly and does not contain any point from \( \{p, q\} \). At least one disk remains unstabbed.

\[ \square \]

### Conclusion

We gave a simple linear-time algorithm to find five stabbing points for a set of pairwise intersecting disks in the plane. It remains open how to use the proofs of Danzer or Stachó [15, 5] (or any other technique) for an efficient construction of four stabbing points. It is also not known whether nine disks can be stabbed by three points or not (for eight disks, this is the case [14]). Furthermore, it would be interesting to find a simpler construction, than the one by Danzer, of ten pairwise intersecting disks that cannot be stabbed by three points.

### References

Point Location in Incremental Planar Subdivisions

Eunjin Oh
Max Planck Institute for Informatics, Saarbrücken, Germany
eoh@mpi-inf.mpg.de

Abstract
We study the point location problem in incremental (possibly disconnected) planar subdivisions, that is, dynamic subdivisions allowing insertions of edges and vertices only. Specifically, we present an $O(n \log n)$-space data structure for this problem that supports queries in $O(\log^2 n)$ time and updates in $O(\log n \log \log n)$ amortized time. This is the first result that achieves polylogarithmic query and update times simultaneously in incremental planar subdivisions. Its update time is significantly faster than the update time of the best known data structure for fully-dynamic (possibly disconnected) planar subdivisions.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases Dynamic point location, general incremental planar subdivisions

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.51

Related Version A full version of this paper is available at http://arxiv.org/abs/1809.10495.

1 Introduction

Given a planar subdivision, a point location query asks for finding the face of the subdivision containing a given query point. The planar subdivisions for point location queries are induced by planar embeddings of graphs. A planar subdivision consists of faces, edges and vertices whose union coincides with the whole plane. An edge of a subdivision is considered to be open, that is, it does not include its endpoints (vertices). A face of a subdivision is a maximal connected subset of the plane that does not contain any point on an edge or a vertex. The boundary of a face of a subdivision may consist of several connected components. Imagine that we give a direction to each edge on the boundary of a face $F$ so that $F$ lies to the left of it. (If an edge is incident to $F$ only, we consider it as two edges with opposite directions.) We call a boundary component of $F$ the outer boundary of $F$ if it is traversed in counterclockwise order around $F$. Every bounded face has exactly one outer boundary. We call a connected component other than the outer boundary an inner boundary of $F$.

We say a planar subdivision is dynamic if the subdivision changes dynamically by insertions and deletions of edges and vertices. A dynamic planar subdivision is connected if the underlying graph is connected at any time. In other words, the boundary of each face is connected. We say a dynamic planar subdivision is general if it is not necessarily connected. There are three versions of dynamic planar subdivisions with respect to the update operations they support: incremental, decremental and fully-dynamic. An incremental subdivision allows only insertions of edges and vertices, and a decremental subdivision allows only deletions of edges and vertices. A fully-dynamic subdivision allows both of them.

The dynamic point location problem is closely related to the dynamic vertical ray shooting problem in the case of connected subdivisions [6]. In this problem, we are asked to find the edge of a dynamic planar subdivision that lies immediately above a query point. The boundary of each face in a dynamic connected subdivision is connected, so one can maintain
the boundary of each face efficiently using a concatenable queue. Then one can answer a point location query without increasing the space and time complexities using a data structure for the dynamic vertical ray shooting problem [6].

However, it is not the case in general planar subdivisions. Although the dynamic vertical ray shooting data structures presented in [1, 2, 4, 6] work for general subdivisions, it is unclear how one can use them to support point location queries efficiently. As pointed out in previous works [4, 6], a main issue concerns how to test for any two edges if they belong to the boundary of the same face in the subdivision. This is because the boundary of a face may consist of more than one connected component.

Previous work. There are several data structures for the point location problem in fully-dynamic planar connected subdivisions [1, 2, 4, 6, 7, 8, 10, 14]. The latest result was given by Chan and Nekrich [4]. The linear-size data structure by Chan and Nekrich [4] supports $O((\log n (\log \log n)^2)$ query time and $O((\log n \log \log n)$ update time in the pointer machine model, where $n$ is the number of the edges of the current subdivision. Some of them [1, 2, 4, 6] including the result by Chan and Neckrich can be used for answering vertical ray shooting queries without increasing the running time.

There are data structures for answering point location queries more efficiently in incremental planar connected subdivisions in the pointer machine model [1, 10, 11]. The best known data structure supports $O((\log n \log^* n)$ query time and $O((\log n)$ amortized update time, and it has size of $O(n)$ [1]. This data structure can be modified to support $O((\log n)$ query time and $O((\log^1+\epsilon n)$ amortized update time for any $\epsilon > 0$. In the case that every cell is monotone at any time, there is a linear-size data structure supporting $O((\log n \log \log n)$ query time and $O(1)$ amortized update time [10].

On the other hand, little has been known about this problem in fully-dynamic planar general subdivisions, which was recently mentioned by Snoeyink [15]. Very recently, Oh and Ahn [13] presented a linear-size data structure for answering point location queries in $O((\log n (\log \log n)^2)$ time with $O(\sqrt{n} \log n (\log \log n)^{3/2})$ amortized update time. In fact, this is the only data structure known for answering point location queries in general dynamic planar subdivisions. In the same paper, the authors also considered the point location problem in decremental general subdivisions. They presented a linear-size data structure supporting $O((\log n)$ query time and $O(\alpha(n))$ update time, where $n$ is the number of edges in the current subdivision and $\alpha(n)$ is the inverse Ackermann function.

Our result. In this paper, we present a data structure for answering point location queries in incremental general planar subdivisions in the pointer machine model. The data structure supports $O((\log^2 n)$ query time and $O((\log n \log \log n)$ amortized update time. This is the first result on the point location problem specialized in incremental general planar subdivisions. The update time of this data structure is significantly faster than the update time of the data structure in fully-dynamic general planar subdivisions in [13].

Comparison to the decremental case. In decremental general subdivisions, there is a simple and efficient data structure for point location queries [13]. This data structure maintains the decremental subdivision explicitly: for each face $F$ of the subdivision, it maintains a number of (concatenable) queues each of which stores the edges of each connected component of the boundary of $F$. When an edge is removed, two faces might be merged into one face, but no face is subdivided into two faces. Using this property, they maintain a disjoint-set data structure for each face such that an element of the disjoint-set data structure is the name of a queue representing a connected component of the boundary of this face.
In contrast to decremental subdivisions, it is unclear how to maintain incremental subdivisions explicitly. Suppose that a face $F$ is subdivided into $F_1$ and $F_2$ by the insertion of an edge $e$. An inner boundary of $F$ becomes an inner boundary of either $F_1$ or $F_2$ after $e$ is inserted. See Figure 1(a). It is unclear how to update the set of the inner boundaries of $F_i$ for $i = 1, 2$ without accessing every queue representing an inner boundary of $F$. If we access all such queues, the total insertion time for $n$ insertions is $\Omega(n^2)$ in the worst case. Therefore it does not seem that the approach in [13] works for incremental subdivisions.

### 2 Preliminaries

Consider an incremental planar subdivision $\Pi$. We use $\overline{\Pi}$ to denote the union of the edges and vertices of $\Pi$. We require that every edge of $\Pi$ be a straight line segment. For a set $A$ of elements (points or edges), we use $|A|$ to denote the number of the elements in $A$. For a planar subdivision $\Pi'$, we use $|\Pi'|$ to denote the complexity of $\Pi'$, i.e., the number of the edges of $\Pi'$. We use $n$ to denote the number of the edges of $\Pi$ at the moment. Also, for a connected component $\gamma$ of $\Pi$, we use $\Pi_\gamma$ to denote the subdivision induced by $\gamma$. Notice that $\Pi_\gamma$ is connected. Due to lack of space, proofs and details are omitted. Missing proofs and details can be found in the full version of this paper.

In this problem, we are to process a mixed sequence of $n$ edge insertions and vertex insertions so that given a query point $q$ the face of the current subdivision containing $q$ can be computed efficiently. More specifically, each face in the subdivision is assigned a distinct name, and given a query point the name of the face containing the point is to be reported. For the insertion of an edge $e$, we require $e$ to intersect no edge or vertex in the current subdivision. Also, an endpoint of $e$ is required to lie on a face or a vertex of the subdivision. We insert the endpoints of $e$ in the subdivision as vertices if they were not vertices of the subdivision. For the insertion of a vertex $v$, it lies on an edge or a face of the current subdivision. If it lies on an edge, the edge is split into two (sub)edges whose common endpoint is $v$.

### 2.1 Tools

In this subsection, we introduce tools we use. A *concatenable queue* represents a sequence of $N$ elements, and allows five operations: insert an element, delete an element, search an element, split a queue into two queues, and concatenate two queues into one. By implementing them with 2-3 trees, we can support each operation in $O(\log N)$ time.
The vertical decomposition of a (static) planar subdivision $\Pi_s$ is a finer subdivision of $\Pi_s$ induced by vertical line segments. For each vertex $v$ of $\Pi_s$, consider two vertical extensions from $v$, one going upwards and one going downwards. The extensions stop when they meet an edge of $\Pi_s$ other than the edges incident to $v$. The vertical decomposition of $\Pi_s$ is the subdivision induced by the vertical extensions contained in the bounded faces of $\Pi_s$ together with the edges of $\Pi_s$. Note that the unbounded face of $\Pi_s$ remains the same. In this paper, we do not consider the unbounded face of $\Pi_s$ as a cell of the vertical decomposition. Therefore, every cell is a trapezoid or a triangle (a degenerate trapezoid). There are $O(|\Pi_s|)$ trapezoids in the vertical decomposition of $\Pi_s$. We treat each trapezoid as a closed set. We can compute the vertical decomposition in $O(|\Pi_s|)$ time [5] since we do not decompose the unbounded face of $\Pi_s$.

We use segment trees, interval trees and priority search trees as basic building blocks. In the following, we briefly review those trees. But we use priority search trees and interval trees of larger fan-out only in the part omitted in the main text, so we also omit their description. Their description can be found in the full version of this paper. For more information, refer to [9, Section 10].

We first introduce the segment tree and the interval tree on a set $I$ of $n$ intervals on the $x$-axis. Let $I_p$ be the set of the endpoints of the intervals of $I$. The base tree is a binary search tree on $I_p$ of height $O(\log n)$ such that each leaf node corresponds to exactly one point of $I_p$. Each internal node $v$ corresponds to a point $\ell(v)$ on the $x$-axis and an interval $\pi(v)$ on the $x$-axis such that $\ell(v)$ is the midpoint of $\pi(v) \cap \text{region}(v)$. For the root $v$, $\text{region}(v)$ is defined as the $x$-axis. Suppose that $\ell(v)$ and $\text{region}(v)$ are defined for a node $v$. For its two children $v_l$ and $v_r$, $\text{region}(v_l)$ and $\text{region}(v_r)$ are the left and right regions of $\text{region}(v)$, respectively, in the subdivision of $\text{region}(v)$ induced by $\ell(v)$.

For the interval tree, each interval $I \in I$ is stored in exactly one node: the node $v$ of maximum depth with $\text{region}(v) \subseteq I$, that is, the lowest common ancestor of two leaf nodes corresponding to the endpoints of $I$. For the segment tree, each interval $I$ is stored in $O(\log n)$ nodes: the nodes $v$ with $\text{region}(v) \subseteq I$, but $\text{region}(u) \not\subseteq I$ for the parent $u$ of $v$. For any point $p \in \mathbb{R}$, let $\pi(p)$ be the search path of $p$. The intervals of $I$ containing $p$ are stored in some nodes of $\pi(p)$ in both trees. However, not every interval stored in such nodes contains $p$ in the interval tree while every interval stored in such nodes contains $p$ in the segment tree.

Similarly, the segment tree and the interval tree on a set $S$ of $n$ line segments in the plane are defined as follows. Let $S_x$ be the set of the projections of the line segments of $S$ onto the $x$-axis. The segment and interval trees of $S$ are basically the segment and interval trees on $S_x$, respectively. The only difference is that instead of storing the projections, we store a line segment of $S$ in the nodes where its projection is stored in the case of $S_x$. As a result, $\ell_x(v)$ and $\text{region}_x(v)$ for the trees of $S$ are naturally defined as the vertical line containing $\ell(v)$ and the smallest vertical slab containing $\text{region}(v)$ for the trees of $S_x$, respectively. If it is clear in context, we use $\ell(v)$ and $\text{region}(v)$ to denote $\ell_x(v)$ and $\text{region}_x(v)$, respectively.

### 2.2 Subproblem: Stabbing-Lowest Query Problem for Trapezoids

The trapezoids we consider have two sides parallel to the $y$-axis. We consider the stabbing-lowest query problem for trapezoids as a subproblem. In this problem, we are given a set $T$ of trapezoids which is initially empty and changes dynamically by insertions of trapezoids. Here, the trapezoids we are given satisfy that no two upper or lower sides of the trapezoids cross each other. But it is possible that the upper (or lower) side of one trapezoid crosses a vertical side of some other trapezoid. We process a sequence of updates so that given a query point $q$, the trapezoid with lowest upper side can be found efficiently among all trapezoids.
of $\mathcal{T}$ containing $q$. Here, we say a trapezoid has the \textit{lowest} upper side if its upper side is intersected first by the vertical upward ray from $q$ among all upper sides of the trapezoids of $\mathcal{T}$ containing $q$. We call such a trapezoid the \textit{lowest trapezoid stabbed by $q$}.

In Section 4, we present a data structure for this problem. The worst-case query time is $O(\log^2 n)$, the amortized update time is $O(\log n \log \log n)$, and the size of the data structures is $O(n \log n)$. We will use this data structure as a black box in Section 3.

3 Point Location in Incremental General Planar Subdivisions

Compared to connected subdivisions, a main difficulty for handling dynamic general planar subdivisions lies in finding the faces incident to the edge $e$ lying immediately above a query point [6]. If $e$ is contained in the outer boundary of a face, we can find such a face as the algorithm in [6] for connected planar subdivisions does. However, this approach does not work if $e$ lies on an inner boundary of a face. To overcome this difficulty, instead of finding the edge in $\Pi$ lying immediately above a query point $q$, we find an outer boundary edge of the face $F$ of $\Pi$ containing $q$. See Figure 1(b). To do this, we answer a point location query in two steps.

First, we find the (maximal) connected component $\gamma$ of $\overline{\Pi}$ containing the outer boundary of the face $F$ containing the query point $q$. We use $\text{FindCC}(\Pi)$ to denote this data structure. Observe that the boundary of the face of $\overline{\Pi}$ containing $q$ coincides with the outer boundary of $F$. We maintain the boundary of each face of $\overline{\Pi}$ using a concatenable queue. Thus given an outer boundary edge of $F$, we can return the name of $F$ by defining the name of each face of $\overline{\Pi}$ as the name of the concatenable queue representing its outer boundary.

Second, we apply a point location query on $\overline{\Pi}_\gamma$. More specifically, we find the face $F_\gamma$ in $\overline{\Pi}_\gamma$ containing $q$, find the concatenable queue representing the boundary of $F_\gamma$, and return its name. Since $\overline{\Pi}_\gamma$ is connected, we can maintain an efficient data structure for point location queries on $\overline{\Pi}_\gamma$. We use $\text{LocateCC}(\gamma)$ to denote this data structure. Each of Sections 3.1 and 3.2 describes each of the two data structures together with query and update algorithms.

In addition to them, we maintain the following data structures: one for checking if a new edge is incident to $\overline{\Pi}$, one for maintaining the connected components of $\overline{\Pi}$, and one for maintaining the concatenable queue for the outer boundary of each face of $\Pi$. Details can be found in the full version.

3.1 FindCC(\Pi): Finding One Connected Component for a Query Point

We construct a data structure for finding the (maximal) connected component $\gamma_q$ of $\overline{\Pi}$ containing the outer boundary of the face of $\Pi$ containing a query point $q$. To do this, we compute a set $\mathcal{T}$ of $O(n)$ trapezoids each of which \textit{belongs} to exactly one edge of $\Pi$ such that the edge to which the lowest trapezoid stabbed by $q$ belongs is contained in $\gamma_q$. Then we construct the stabbing-lowest data structure on $\mathcal{T}$ described in Section 4.

Data structure and query algorithm. For each connected component $\gamma$ of $\overline{\Pi}$, consider the subdivision $\Pi_\gamma$ induced by $\gamma$. Notice that $\Pi_\gamma$ is connected. Let $U(\gamma)$ be the union of the closures of all bounded faces of $\Pi_\gamma$. Note that it might be disconnected. Imagine that we have the cells (trapezoids) of the vertical decomposition of $U(\gamma)$. Note that an edge of $\gamma$ might intersect a cell. We say that a cell of the decomposition \textit{belongs} to the edge of $\gamma$ containing the upper side of the cell. Let $\mathcal{T}_\gamma$ be the set of such cells (trapezoids) for $\gamma$, and $\mathcal{T}$ be the union of $\mathcal{T}_\gamma$ for every connected component $\gamma$ of $\overline{\Pi}$. See Figure 2. In the full version, we show that the lowest trapezoid in $\mathcal{T}$ stabbed by a query point $q$ belongs to an edge in $\gamma_q$. If no trapezoid in $\mathcal{T}$ contains $q$, we conclude that $q$ is contained in the unbounded face of $\Pi$. 

ISAAC 2018
Figure 2 (a) The component $\gamma$ contains the outer boundary of the face containing $q$. (b) Using the vertical decomposition, we obtain $O(n)$ (possibly intersecting) trapezoids. Their corners are marked with disks. The lowest trapezoid stabbed by $q$ is the dashed one, which comes from $\gamma$.

However, each edge insertion may induce $\Omega(n)$ changes on $T$ in the worst case. For an efficient update procedure, we define and construct the trapezoid set $T_\gamma$ in a slightly different way by allowing some edges lying inside $U(\gamma)$ to define trapezoids in $T_\gamma$. For a connected component $\gamma$ of $\Pi$, we say a set of connected subdivisions induced by edges of $\gamma$ covers $\gamma$ if an edge of $\gamma$ is contained in at most two subdivisions, and one of the subdivisions contains all edges of the boundary of $U(\gamma)$. Let $\mathcal{F}_\gamma$ be a set of connected subdivisions covering $\gamma$. See Figure 3. Notice that $\mathcal{F}_\gamma$ is not necessarily unique. For a technical reason, if the union of some edges (including their endpoints) in a subdivision of $\mathcal{F}_\gamma$ forms a line segment, we treat them as one edge. Then we let $T_\gamma$ be the set of the cells of the vertical decompositions of the subdivisions in $\mathcal{F}_\gamma$. Note that a cell of $T_\gamma$ might intersect another cell of $T_\gamma$. See Figure 3(b). We say that a cell (trapezoid) of $T_\gamma$ belongs to the edge of $\gamma$ containing the upper side of the cell. Let $T$ be the union of all such sets $T_\gamma$.

Due to the following lemma, we can maintain $T$ efficiently. In the update algorithm, we insert trapezoids to $T$ only.

**Lemma 1.** The size of $T$ is $O(n)$, where $n$ is the complexity of the current subdivision.

The following lemma shows that the lowest trapezoid in $T$ stabbed by $q$ belongs to an edge of $\gamma_q$. Thus by constructing a stabbing-lowest data structure on $T$, we can find $\gamma_q$ in $O(Q(n))$ time, where $Q(n)$ is the query time for answering a stabbing-lowest query. The query time of the structure on $n$ trapezoids described in Section 4 is $O(\log^2 n)$.

**Lemma 2.** The lowest trapezoid in $T$ stabbed by a query point $q$ belongs to an edge of the connected component of $\Pi$ containing the outer boundary of the face of $\Pi$ containing $q$. If the face of $\Pi$ containing $q$ is unbounded, no trapezoid in $T$ contains $q$.

**Lemma 3.** Given $\text{FindCC}(\Pi)$ of size $O(n)$, we can find the connected component of $\Pi$ containing the outer boundary of the face of $\Pi$ containing a query point in $O(\log^2 n)$ time.

**Update algorithm.** We maintain a stabbing-lowest data structure on $T$. Let $T_\gamma$ be the set of the trapezoids of $T$ which belong to edges of $\gamma$. Notice that we do not maintain the sets $\mathcal{F}_\gamma$ and $T_\gamma$ for a connected component $\gamma$ of $\Pi$. We use them only for description purpose. Here, we describe the update algorithm for the insertion of an edge only. The update algorithm for the insertion of a vertex can be found in the full version.

We process the insertion of an edge $e$ by inserting a number of trapezoids to $T$. Here, we use $\Pi$ to denote the subdivision of complexity $n$ before $e$ is inserted. There are four cases: $e$ is not incident to $\Pi$, only one endpoint of $e$ is contained in $\Pi$, the endpoints of $e$ are contained in distinct connected components of $\Pi$, and the endpoints of $e$ are contained in the same
connected component of \( \Pi \). We can check if \( e \) belongs to each case in \( O(\log n) \) time using the data structure described at the beginning of Section 3 in the full version. For the first three cases, we do not need to update \( T \). This is because no new face appears in the current subdivision. Thus the conditions on the definition of \( F_\gamma \) are not violated in these cases. (We will see this in more detail in the proof of Lemma 4.)

Now consider the remaining case: the endpoints of \( e \) are contained in the same connected component, say \( \gamma \), of \( \Pi \). Recall that \( U(\gamma) \) is closed. If \( e \) is contained in the interior of \( U(\gamma) \), we do nothing since \( F_\gamma \) covers \( \gamma \cup e \). We can check this in constant time. Details can be found in the full version. If \( e \) is not contained in the interior of \( U(\gamma) \), we trace the edges of the new face in time linear in the complexity of the new face using the data structures presented at the beginning of Section 3 in the full version. Then we compute the vertical decomposition of the face in the same time \([5]\), and insert them to \( T \). This takes time linear in the number of the new trapezoids inserted to \( T \), which is \( O(n) \) in total over all updates by Lemmas 1 and 4, and the fact that no trapezoid is removed from \( T \). As new trapezoids are inserted to \( T \), we update the stabbing-lowest data structure on \( T \).

For the correctness, we have the following lemma. A proof can be found in the full version.  

\( \blacktriangleright \) **Lemma 4.** For each connected component \( \gamma \) of \( \Pi \), there is a set \( F_\gamma \) of connected subdivisions covering \( \gamma \) such that \( T_\gamma \) consists of the cells of the vertical decompositions of the subdivisions of \( F_\gamma \) at any moment.

Let \( S(n), Q(n) \) and \( U(n) \) be the size, the query time and the update time of an insertion-only stabbing-lowest data structure for \( n \) trapezoids, respectively. In the case of the data structure described in Section 4, we have \( S(n) = O(n \log n) \), \( Q(n) = O(\log^2 n) \) and \( U(n) = O(\log n \log \log n) \). Recall that the total number of trapezoids inserted to \( T \) is \( O(n) \). We have the following lemma.  

\( \blacktriangleright \) **Lemma 5.** We can construct a data structure of size \( O(S(n)) \) so that the connected component of \( \Pi \) containing the outer boundary of the face containing \( q \) can be found in \( O(Q(n)) \) worst case time for any point \( q \) in the plane, where \( n \) is the number of edges at the moment. Each update takes \( O(U(n)) \) amortized time.

3.2 LocateCC(\( \gamma \)): Find the Face Containing a Query Point in \( \Pi_\gamma \)  

For each connected component \( \gamma \) of \( \Pi \), we maintain a data structure, which is denoted by LocateCC(\( \gamma \)), for finding the face of \( \Pi_\gamma \) containing a query point. Here, we need two update operations for LocateCC(\( \cdot \)): inserting a new edge to LocateCC(\( \cdot \)) and merging two data structures LocateCC(\( \gamma_1 \)) and LocateCC(\( \gamma_2 \)) for two connected components \( \gamma_1 \) and \( \gamma_2 \) of \( \Pi \). Notice that we do not need to support edge deletion since \( \Pi \) is incremental.
No known point location data structure supports the merging operation explicitly. Instead, one simple way is to make use of the edge insertion operation which is supported by most of the known point location data structures. For merging two data structures, we simply insert every edge in the connected component of smaller size to the data structure for the other connected component. By using a simple charging argument, we can show that the amortized update time (insertion and merging) is \( O(U'(n) \log n) \), where \( U'(n) \) is the insertion time of the dynamic point location data structure we use. If we use the data structure by Arge et al. [1], the query time is \( O(\log n \log^* n) \) and the amortized update time is \( O(\log^2 n) \).

In this section, we improve the update time at the expense of increasing the query time. Because \textsc{FindCC}(\Pi)\hspace{1pt}\textsc{LocateCC}\hspace{1pt}\textsc{Data structure and query algorithm} requires \( O(\log^2 n) \) query time, we are allowed to spend more time on a point location query on \( \gamma \). The data structure proposed in this section supports \( O(\log^2 n) \) query time. The amortized update time is \( O(\log n \log \log n) \).

**Data structure and query algorithm.** \textsc{LocateCC}(\gamma) allows us to find the face of \( \Pi_\gamma \) containing a query point. Since \( \gamma \) is connected and we maintain the outer boundary of each face of \( \Pi \), it suffices to construct a vertical ray shooting structure for the edges of \( \gamma \). Recall that the boundary of a face of \( \Pi_\gamma \) coincides with the outer boundary of a face of \( \Pi \). The vertical ray shooting problem is \textit{decomposable} in the sense that we can answer a query on \( S_1 \cup S_2 \) in constant time once we have the answers to queries on \( S_1 \) and \( S_2 \) for any two sets \( S_1 \) and \( S_2 \) of line segments in the plane. Thus we can use an approach by Bentley and Saxe [3].

We decompose the edge set of \( \gamma \) into subsets of distinct sizes such that each subset consists of exactly \( 2^i \) edges for some index \( i \leq \lceil \log n \rceil \). Note that there are \( O(\log n) \) subsets in the decomposition. We use \( \mathcal{B}(\gamma) \) to denote the set of such subsets, and \( \mathcal{B} \) to denote the union of \( \mathcal{B}(\gamma) \) for all connected components \( \gamma \) of \( \Pi \). \textsc{LocateCC}(\gamma) consists of \( O(\log n) \) static vertical ray shooting data structures, one for each subset in \( \mathcal{B}(\gamma) \). To answer a query on \( \gamma \), we apply a vertical ray shooting query on each subset of \( \mathcal{B}(\gamma) \), and choose the one lying immediately above the query point. This takes \( O(Q_s(n) \log n) \) time, where \( Q_s(n) \) denotes the query time of the static vertical ray shooting data structure we use.

For a static vertical ray shooting data structure \( \mathcal{D}_s(\beta) \) for \( \beta \in \mathcal{B} \), we present a variant of the (dynamic) vertical ray shooting data structure of Arge et al. [1]. It supports \( O(\log n) \) query time, and an efficient merging operation. In the update procedure, we merge two subsets \( \beta_1 \) and \( \beta_2 \) in \( \mathcal{B} \) into one, and merge their static vertical ray shooting data structures. If we construct \( \mathcal{D}_s(\beta_1 \cup \beta_2) \) from scratch, the total update time is \( \Omega(n \log^2 n) \) because the construction of a vertical ray shooting data structure on \( N \) segments takes \( \Omega(N \log N) \) time for any data structure. To improve this update time, we maintain a set of sorted lists of edges, which we call the backbone tree, so that we can merge two static ray shooting data structures more efficiently. Notice that the edges of \( \Pi \) cannot be consistently sorted with respect to the \( y \)-axis in advance. This happens if no vertical line crosses two edges of \( \Pi \). The \( y \)-order of the two edges depends on the edges to be inserted. In our case, we maintain sets of edges which can be consistently sorted (i.e., edges intersecting a common vertical line), and maintain their sorted lists. Details can be found in the full version. Proofs of the following lemmas can also be found in the full version.

- **Lemma 6.** Given \( \mathcal{D}_s(\beta) \) for every subset \( \beta \in \mathcal{B} \), we can find the edge lying immediately above a query point among the edges of a connected component \( \gamma \) of \( \Pi \) in \( O(\log^2 n) \) time.

- **Lemma 7.** Given \( \mathcal{D}_s(\beta_1) \) and \( \mathcal{D}_s(\beta_2) \) for two subsets \( \beta_1 \) and \( \beta_2 \) of \( \mathcal{B} \), we can construct \( \mathcal{D}_s(\beta) \) in \( O(|\beta| \log \log n) \) time, where \( \beta = \beta_1 \cup \beta_2 \).
Update algorithm. We have two update operations, the insertion of edges and vertices. We do not need to update LOCATECC(·) in the case of a vertex insertion. Details can be found in the full version. We use II to denote the subdivision of complexity \( n \) before \( e \) is inserted.

Suppose that we are given an edge \( e \) and we are to update LOCATECC(·). Specifically, we update the static vertical ray shooting data structures for some subsets of \( B \) and the backbone tree. We find the connected components of \( \Pi \) incident to \( e \) in \( O(\log n) \) time. There are three cases: there is no such connected component, there is only one such connected component, or there are two such connected components. We show how to update the data structure only for the last case. Details for the other cases can be found in the full version.

For the last case, let \( \gamma_1 \) and \( \gamma_2 \) be two connected components incident to \( e \). They are merged into one connected component together with \( e \). If every subset in \( B(\gamma_1) \) and \( B(\gamma_2) \) has distinct size, we just collect every static vertical ray shooting data structure constructed on a subset in \( B(\gamma_1) \cup B(\gamma_2) \), and insert \( e \) to the data structure. Then we are done. If not, we first choose the largest subsets, one from \( B(\gamma_1) \) and one from \( B(\gamma_2) \), of the same size, say \( 2^i \). Then we construct a new vertical ray shooting data structure on the union \( \beta' \) of the two subsets in \( O(2^{i+1}\log \log n) \) time. If there is a subset in \( B(\gamma_1) \) or \( B(\gamma_2) \) of size \( 2^{i+1} \) other than \( \beta' \), we again merge them together to form a subset of size \( 2^{i+2} \). We repeat this until every subset in \( B(\gamma_1) \) and \( B(\gamma_2) \) of size at least \( 2^i \) has distinct size. Then we consider the largest subsets, one from \( B(\gamma_1) \) and one from \( B(\gamma_2) \), of the same size again. Note that the size of the two subsets is less than \( 2^i \). We merge them, and repeat the merge procedure. We do this for every pair of subsets in \( B(\gamma_1) \) and \( B(\gamma_2) \) of the same size. Finally, we have the set \( B(\gamma_1 \cup \gamma_2) \) of subsets of the edges of \( \gamma_1 \cup \gamma_2 \) of distinct sizes, and the static vertical ray shooting data structure for each subset in \( B(\gamma_1 \cup \gamma_2) \). Then we insert \( e \) to the data structure. Details can be found in the full version.

Lemma 8. The total time for updating every vertical ray shooting data structure in the course of \( n \) edge insertions is \( O(n \log n \log \log n) \).

Lemma 9. We can maintain a data structure of size \( O(n \log \log n) \) in an incremental planar subdivision \( \Pi \) so that the edge of \( \gamma \) lying immediately above \( q \) can be found in \( O(\log^2 n) \) time for any edge \( e \) and any connected component \( \gamma \) of \( \Pi \). The amortized update time of this data structure is \( O(\log n \log \log n) \).

4 Incremental Stabbing-Lowest Data Structure for Trapezoids

In this section, we are given a set \( T \) of trapezoids which is initially empty. Then we are to process the insertions of trapezoids to \( T \) so that the lowest trapezoid in \( T \) stabbed by a query point can be found efficiently. Recall that the upper and lower sides of the trapezoids we consider in this paper do not cross each other. To make the description easier, we present a simplified version of our data structure supporting \( O(\log^2 n \log \log n) \) query time and \( O(\log n \log \log n) \) insertion time in the main text. By using an interval tree of fan-out \( \log^2 n \), we can improve the query time by a factor of \( \log \log n \). Details can be found in the full version.

Data structure. The base tree is an interval tree of the upper and lower sides of the trapezoids of \( T \). Since the left and right sides of the trapezoids are parallel to the y-axis, a node of the interval tree stores the upper side of a trapezoid of \( T \) if and only if it stores the lower side of the trapezoid. Here, instead of storing the upper and lower sides of a trapezoid, we store the trapezoid itself in such a node. In this way, a trapezoid \( \square \) of \( T \) is stored in at most one node of the interval tree. For details, refer to Section 2.
We construct a secondary structure associated with a node \( v \) of the base tree as follows. Let \( S(v) \) be the set of the trapezoids stored in \( v \). Every trapezoid of \( S(v) \) intersects a common vertical line \( \ell(v) \). Thus, their upper and lower sides can be sorted in their \( y \)-order. See Figure 4. Let \( I(v) \) be the set of the intersections of the trapezoids of \( S(v) \) with \( \ell(v) \). Note that it is a set of intervals of \( \ell(v) \). We construct a segment tree \( T(v) \) of \( I(v) \). A node \( u \) of \( T(v) \) corresponds to an interval \( \text{region}(u) \) contained in \( \ell(v) \). Every interval of \( I(v) \) stored in \( u \) contains \( \text{region}(u) \). An interval \( I \in I(v) \) has its corresponding trapezoid \( \Box \) in \( S(v) \) such that \( \Box \cap \ell(v) = I \). We let \( I \) have the \textit{key} which is the \( x \)-coordinate of the left side of \( \Box \).

For each node \( u \) of \( T(v) \), we construct a tertiary data structure so that given a query value \( x \) the interval with lowest upper endpoint can be found efficiently among the intervals stored in \( u \) and having their keys less than \( x \). Imagine that we sort the intervals of \( I(v) \) stored in \( u \) with respect to their keys, and denote them by \( (I_1, \ldots, I_k) \). And we use \( \Box \in T \) to denote the trapezoid corresponding to the interval \( I_i \) (i.e., \( \ell(v) \cap \Box = I_i \)) for \( i = 1, \ldots, k \). The tertiary data structure is just a sublist of \( (I_1, \ldots, I_k) \). Specifically, suppose \( x \) is at least the key of \( I_i \), and at most the key of \( I_{i+1} \) for some \( i \). Then every interval in \( (I_1, I_2, \ldots, I_i) \) has its key at most \( x \). Thus the answer to the query is the one with lowest upper endpoint among \( (I_1, I_2, \ldots, I_i) \). Using this observation, we construct a sublist of \( (I_1, \ldots, I_k) \) as follows. We choose the interval, say \( I_i \), if its upper endpoint is the lowest among the upper endpoints of the intervals in \( (I_1, \ldots, I_i) \). We maintain the sublist consisting of the chosen intervals. Notice that the sublist has \textit{monotonicity} with respect to their upper endpoints. That is, the upper endpoint of \( I_i \) lies lower than the upper endpoint of \( I_i' \) if \( I_i \) comes before \( I_i' \) in the sublist. This property makes the update procedure efficient.

By applying binary search on the sublist with respect to the keys, we can find the interval with lowest endpoint among the intervals stored in \( u \) and having the keys less than \( x \). For each node of the base tree, we maintain a structure for dynamic fractional cascading [12] on the segment tree so that the binary search on the sublist associated with each node of the segment tree can be done in \( O(\log n \log \log n) \) time in total. Then we also do this for the right sides of the trapezoids of \( S(v) \).

A tricky problem here is that a query point \( q \) and the upper or lower side of a trapezoid in \( S(v) \) cannot be ordered with respect to the \( y \)-axis in general. This happens if the left side of the trapezoid lies to the right of \( q \). See Figure 4. This makes it difficult to follow a search path in the segment tree associated with \( v \). To resolve this, we find the side \( e \) lying immediately above \( q \) among the upper and lower sides of the trapezoids in \( S(v) \), and then follow the search path of \( q' = e \cap \ell(v) \). To do this, we construct a vertical ray shooting data structure on the upper and lower sides of the trapezoids in \( S(v) \). Details can be found in the full version.
Query algorithm. Using this data structure, we can find the lowest trapezoid in $T$ stabbed by a query point $q$ as follows. We follow the base tree (interval tree) along the search path $\pi$ of $q$ of length $O(\log n)$. For each node of $\pi$, we consider its associated secondary structures, and we find the lowest trapezoid stabbed by $q$ among the trapezoids stored in the node. And we return the lowest one among all trapezoids we obtained from the nodes of $\pi$. We spend $O(\log n \log \log n)$ time on each node in $\pi$, which leads to the total query time of $O(\log^2 n \log \log n)$.

We have a segment tree on the intersections of the trapezoids of $S(v)$ with $\ell(v)$ for a node $v$ in $\pi$. We first find the upper or lower side $e$ of a trapezoid of $S(v)$ immediately lying above $q$ among them in $O(\log n)$ time using the vertical ray shooting data structure associated with $v$, and let $q'$ be the intersection point between $e$ and $\ell(v)$. See Figure 4. We show that the lowest trapezoid stabbed by $q$ is stored in a node in the search path of $q'$. A proof can be found in the full version. Thus, it suffices to consider $O(\log n)$ nodes $w$ in the segment tree with $q' \in \text{region}(w)$. Then we find the successor of the $x$-coordinate of $q$ on the sublist associated with each such node. By construction, the trapezoid corresponding to the successor is the lowest trapezoid stabbed by $q$ among all trapezoids stored in $w$. Using dynamic fractional cascading, we can find it in $O(\log \log n)$ time for each node after spending $O(\log n)$ time for the initial binary search of only one node in the segment tree. Thus we can find all successors in $O(\log n \log \log n)$ time.

Lemma 10. Using the data structure described in this section, we can find the lowest trapezoid stabbed by a query point in $O(\log^2 n \log \log n)$ time.

Update algorithm. We assume that the trapezoids to be inserted are known in advance so that we can keep the base tree and all segment trees balanced. We can get rid of this assumption with standard technique using weight-balanced B-trees. We show how to do this in the full version. Let $\square$ be a trapezoid to be inserted to the data structure. We find the node $v$ of maximum depth in the base tree such that $\text{region}(v)$ contains $\square$ in $O(\log n)$ time. The trapezoid $\square$ is to be stored only in this node.

We update the secondary structure (segment tree) for $S(v)$ by inserting $\square$. We find the set $W$ of $O(\log n)$ nodes in the segment tree where $\square$ is to be inserted. Each node $w \in W$ is associated with a sorted list $L(w)$ of intervals stored in $w$. We decide if we store $\square \cap \ell(v)$ in $L(w)$. To do this, we find the position for $\square$ in $L(w)$ by applying binary search on $L(w)$ with respect to the key. Here we do this for every node in $W$, and thus we can apply fractional cascading. The key of each interval in the sorted lists is in $\mathbb{R}$. Thus we can apply (dynamic) fractional cascading so that each binary search takes $O(\log \log n)$ time after spending $O(\log n)$ time on the initial binary search on a node of $W$ [12].

Let $\langle I_1, \ldots, I_k \rangle$ be the sorted list of the intervals stored in $w$. The list $L(w)$ is a sublist of this list, say $\langle I_{i_1}, \ldots, I_{i_n} \rangle$. Let $I_{i_j}$ be the predecessor of $\square \cap \ell(v)$. We determine if $\square$ is inserted to the list in constant time: if the upper side of $\square$ lies below the upper side of the trapezoid $\square_{i_{j+1}}$ with $\square_{i_{j+1}} = I_{i_{j+1}} \cap \ell(v)$, we insert $\square \cap \ell(v)$ to the list. Otherwise, the list stored in $w$ remains the same. If we insert $\square \cap \ell(v)$ to the list, we check if it violates the monotonicity of $L(w)$. To do this, we consider the trapezoid $\square'$ whose corresponding interval lies before $\square$ one by one from $\square_{i_j}$. If the upper side of $\square'$ lies above $\square$, we remove $\square'$ from the list. Each insertion into and deletion from $L(w)$ takes $O(\log \log n)$ time [12]. We do this until the upper side of $\square'$ lies below the upper side of $\square$. The total update time for the insertion of $\square$ is $O(\log n + N \log \log n)$, where $N$ is the number of the trapezoids deleted due to $\square$. We show that the sum of $N$ over all $n$ insertions is $O(n \log n)$ in the full version. Thus the amortized update time is $O(\log n \log \log n)$ time.
In the full version, we show how to improve the query time by a factor of $O(\log \log n)$. Therefore, we have the following lemma.

**Lemma 11.** We can maintain an $O(n \log n)$-size data structure on an incremental set of $n$ trapezoids supporting $O(\log n \log \log n)$ amortized update time so that given a query point $q$, the lowest trapezoid stabbed by $q$ can be computed in $O(\log^2 n)$ time.

**References**

Abstract

We consider the problem of testing, for a given set of planar regions $R$ and an integer $k$, whether there exists a convex shape whose boundary intersects at least $k$ regions of $R$. We provide polynomial-time algorithms for the case where the regions are disjoint axis-aligned rectangles or disjoint line segments with a constant number of orientations. On the other hand, we show that the problem is NP-hard when the regions are intersecting axis-aligned rectangles or 3-oriented line segments. For several natural intermediate classes of shapes (arbitrary disjoint segments, intersecting 2-oriented segments) the problem remains open.

1 M.v.d.K. supported by the Netherlands Organisation for Scientific Research under proj. 628.011.005.
2 M.v.K. supported by the Netherlands Organisation for Scientific Research under proj. 612.001.651.
3 M.L. supported by the Netherlands Organisation for Scientific Research under proj. 614.001.504.
4 F.S. supported by the Netherlands Organisation for Scientific Research under proj. 612.001.651.
5 J.U. supported by the Netherlands Organisation for Scientific Research under proj. 612.001.651.
6 J.V. supported by the Netherlands Organisation for Scientific Research under proj. 612.001.651.
7 L.W. supported by the Minsit. of Research, Techn. and High. Ed. of Indonesia (No. 138.41/E4.4/2015).
Convex Partial Transversals of Planar Regions

Figure 1 (a) A set of 12 regions. (b, c) A convex partial transversal of size 10.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases computational geometry, algorithms, NP-hardness, convex transversals

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.52

Related Version A full version of the paper is available at [9], https://arxiv.org/abs/1809.10078.

1 Introduction

A set of points $Q$ in the plane is said to be in convex position if for every point $q \in Q$ there is a halfplane containing $Q$ that has $q$ on its boundary. Now, let $R$ be a set of $n$ regions in the plane. We say that $Q$ is a partial transversal of $R$ if there exists an injective map $f : Q \rightarrow R$ such that $q \in f(q)$ for all $q \in Q$; if $f$ is a bijection we call $Q$ a full transversal. In this paper, we are concerned with the question whether a given set of regions $R$ admits a convex partial transversal $Q$ of a given cardinality $|Q| = k$. Figure 1 shows an example.

The study of convex transversals was initiated by Arik Tamir at the Fourth NYU Computational Geometry Day in 1987, who asked “Given a collection of compact sets, can one decide in polynomial time whether there exists a convex body whose boundary intersects every set in the collection?” Note that this is equivalent to the question of whether a convex full transversal of the sets exists: given the convex body, we can place a point of its boundary in every intersected region; conversely, the convex hull of a convex transversal forms a convex body whose boundary intersects every set. In 2010, Arkin et al. [2] answered Tamir’s original question in the negative (assuming $P \neq NP$): they prove that the problem is NP-hard, even when the regions are (potentially intersecting) line segments in the plane, regular polygons in the plane, or balls in $\mathbb{R}^3$. On the other hand, they show that Tamir’s problem can be solved in polynomial time when the regions are disjoint segments in the plane and the convex body is restricted to be a polygon whose vertices are chosen from a given discrete set of (polynomially many) candidate locations. Goodrich and Snoeyink [6] show that for a set of parallel line segments, the existence of a convex transversal can be tested in $O(n \log n)$ time. Schlipf [13] further proves that the problem of finding a convex stabber for a set of disjoint bends (that is, shapes consisting of two segments joined at one endpoint) is also NP-hard. She also studies the optimisation version of maximising the number of regions stabbed by a convex shape; we may re-interpret this question as finding the largest $k$ such that a convex partial transversal of cardinality $k$ exists. She shows that this problem is also NP-hard for a set of (potentially intersecting) line segments in the plane.

Related work. Computing a partial transversal of maximum size arises in wire layout applications [14]. When each region in $R$ is a single point, our problem reduces to determining whether a point set $P$ has a subset of cardinality $k$ in convex position. Eppstein et al. [4] solve this in $O(kn^3)$ time and $O(kn^2)$ space using dynamic programming; the total number of convex $k$-gons can also be tabulated in $O(kn^3)$ time [12, 10].
Table 1 New and known results. The arrows indicate that one result is implied by another.

<table>
<thead>
<tr>
<th>line segments:</th>
<th>disjoint</th>
<th>intersecting</th>
</tr>
</thead>
<tbody>
<tr>
<td>parallel 2-oriented</td>
<td>$O(n^k)$ (upper hull only: $O(n^2)$)</td>
<td>N/A</td>
</tr>
<tr>
<td>3-oriented</td>
<td>open</td>
<td>NP-hard</td>
</tr>
<tr>
<td>$\rho$-oriented arbitrary</td>
<td>polynomial</td>
<td>open</td>
</tr>
</tbody>
</table>

rectangles: squares rectangles polynomial open NP-hard [2]

other: bends NP-hard [13] ←

If we allow reusing elements, our problem becomes equivalent to so-called covering color classes introduced by Arkin et al. [1]. Arkin et al. show that for a set of regions $\mathcal{R}$ where each region is a set of two or three points, computing a convex partial transversal of $\mathcal{R}$ of maximum cardinality is NP-hard. Conflict-free coloring has been studied extensively, and has applications in, for instance, cellular networks [5, 7, 8].

Results. Despite the large body of work on convex transversals and natural extensions of partial transversals that are often mentioned in the literature, surprisingly, no positive results were known. We present the first positive results: in Section 2 we show how to test whether a set of parallel line segments admits a convex transversal of size $k$ in polynomial time; we extend this result to disjoint segments of a fixed number of orientations and to disjoint axis-aligned rectangles in Section 3. Although the hardness proofs of Arkin et al. and Schlipf do extend to partial convex transversals, we strengthen these results by showing that the problem is already hard when the regions are 3-oriented segments or axis-aligned rectangles (Section 4). Our results are summarized in Table 1.

For ease of terminology, in the remainder of this paper, we will drop the qualifier “partial” and simply use “convex transversal” to mean “partial convex transversal”. Also, for ease of argument, in all our results we test for weakly convex transversals. This means that the transversal may contain three or more colinear points. Missing proofs can be found in the full version of this paper [9].

2 Parallel disjoint line segments

Let $\mathcal{R}$ be a set of $n$ vertical line segments in $\mathbb{R}^2$. We assume that no three endpoints are aligned. Let $\uparrow \mathcal{R}$ and $\downarrow \mathcal{R}$ denote the sets of upper and lower endpoints of the regions in $\mathcal{R}$, respectively, and let $\perp \mathcal{R} = \uparrow \mathcal{R} \cup \downarrow \mathcal{R}$. In Section 2.1 we focus on computing an upper convex transversal – a convex transversal $Q$ in which all points appear on the upper hull of $Q$– that maximizes the number of regions visited. We show that there is an optimal transversal whose strictly convex vertices lie only on bottom endpoints in $\downarrow \mathcal{R}$. This allows us to develop a dynamic programming algorithm that computes such an optimal upper convex transversal in $O(n^2)$ time. In Section 2.2 we prove that there exists an optimal convex transversal whose strictly convex vertices are taken from the set of all endpoints $\perp \mathcal{R}$, and whose leftmost and rightmost vertices are taken from a discrete set of points. This leads to an $O(n^6)$ time dynamic programming algorithm to compute such a transversal.
Figure 2 (a) The definition of $K[v, w]$. The region $\Lambda(v, w)$ is indicated in purple. The segments counted in $I[u, v]$ are shown in red. (b) The case that $K[v, w] = K[v, u]$, where $u$ corresponds to the predecessor slope of $\text{slope}(vw)$. (c) The case that $K[v, w] = K[w, v] + I[w, v]$.

2.1 Computing an upper convex transversal

Let $k^*$ be the maximum number of regions visitable by an upper convex transversal of $\mathcal{R}$.

Lemma 1. Let $U$ be an upper convex transversal of $\mathcal{R}$ that visits $k$ regions. There exists an upper convex transversal $U'$ of $\mathcal{R}$, that visits the same $k$ regions as $U$, and such that the leftmost vertex, the rightmost vertex, and all strictly convex vertices of $U'$ lie on the bottom endpoints of the regions in $\mathcal{R}$.

Proof. Let $U$ be the set of all upper convex transversals with $k$ vertices. Let $U' \in U$ be an upper convex transversal such that the sum of the $y$-coordinates of its vertices is minimal. Assume, by contradiction, that $U'$ has a vertex $v$ that is neither on the lower endpoint of its respective segment nor aligned with its adjacent vertices. Then we can move $v$ down without making the upper hull non-convex. This is a contradiction. Therefore, all vertices in $U'$ are either aligned with their neighbors (and thus not strictly convex), or at the bottom endpoint of a region.

Let $\Lambda(v, w)$ denote the set of bottom endpoints of regions in $\mathcal{R}$ that lie left of $v$ and below the line through $v$ and $w$. See Fig. 2(a). Let $\text{slope}(vw)$ denote the slope of the supporting line of $\overline{vw}$, and observe that $\text{slope}(\overline{vw}) = \text{slope}(\overline{wu})$.

By Lemma 1 there is an optimal upper convex transversal of $\mathcal{R}$ in which all strictly convex vertices lie on bottom endpoints of the segments. Let $K[v, w]$ be the maximum number of regions visitable by an upper convex transversal that ends at a bottom endpoint $v$, and has an incoming slope at $v$ of at least $\text{slope}(vw)$. Note that the second argument $w$ is used only to specify the slope, and $w$ may be left or right of $v$. We have that

$$K[v, w] = \max_{u \in \Lambda(v, w)} \max_{s \in \Lambda(u, v)} K[u, s] + I[u, v],$$

where $I[u, v]$ denotes the number of regions in $\mathcal{R}$ intersected by the segment $\overline{uv}$ (in which we treat the endpoint at $u$ as open, and the endpoint at $v$ as closed). See Fig. 2(a).

Observation 2. Let $v$, $s$, and $t$ be bottom endpoints of segments in $\mathcal{R}$ with $\text{slope}(st) > \text{slope}(sv)$. We have that $K[v, t] \geq K[v, s]$.

Fix a bottom endpoint $v$, and order the other bottom endpoints $w \in \downarrow \mathcal{R}$ in decreasing order of slope $\text{slope}(vw)$. Let $S_v$ denote the resulting order.
\textbf{Lemma 3.} Let \( v \) and \( w \) be bottom endpoints of regions in \( R \), and let \( u \) be the predecessor of \( w \) in \( S_v \), if it exists (otherwise let \( K[v, u] = -\infty \)). We have that

\[
K[v, w] = \begin{cases} 
\max\{1, K[v, u], K[w, v] + I[w, v]\} & \text{if } w_x < v_x, \\
\max\{1, K[v, u]\} & \text{otherwise}.
\end{cases}
\]

Where \( v_x \) denotes the \( x \)-coordinate of a point \( v \). Lemma 3 now suggests a dynamic programming approach to compute the \( K[v, w] \) values for all pairs of bottom endpoints \( v, w \); we process the endpoints \( v \) on increasing \( x \)-coordinate, and for each \( v \), we compute all \( K[v, w] \) values in the order of \( S_v \). To this end, we need to compute (i) the (radial) orders \( S_v \), for all bottom endpoints \( v \), and (ii) the number of regions intersected by a line segment \( \overline{vw} \), for all pairs of bottom endpoints \( u, v \). We show that we can solve both these problems in \( O(n^2) \) time. We then also obtain an \( O(n^2) \) time algorithm to compute \( k^* = \max_{v, w} K[v, w] \).

\textbf{Computing predecessor slopes.} For each bottom endpoint \( v \), we simply sort the other bottom endpoints around \( v \). This can be done in \( O(n^2) \) time in total \cite{11}. We can obtain \( S_v \) by splitting the resulting list into two lists, one with all endpoints left of \( v \) and one with the endpoints right of \( v \), and merging these lists appropriately. This takes \( O(n^2) \) time.

\textbf{Computing the number of intersections.} We use the standard duality transform \cite{3} to map every point \( p = (p_x, p_y) \) to a line \( p^* : y = p_yx - p_y \), and every non-vertical line \( \ell : y = ax + b \) to a point \( \ell^* = (a, -b) \). Consider the arrangement \( A \) formed by the lines \( p^* \) dual to all endpoints \( p \) (both top and bottom) of all regions in \( R \). Observe that all our query segments \( \overline{uw} \) with \( u_x < v_x \) are defined by two bottom endpoints \( u \) and \( v \), so the supporting line \( \ell_{uv} \) of such a segment corresponds to a vertex \( \ell_{uv}^* \) of the arrangement \( A \).

In the dual space, a vertical line segment \( R = \overline{pq} \in R \) corresponds to a strip \( R^* \) bounded by two parallel lines \( p^* \) and \( q^* \). Let \( R^* \) denote this set of strips corresponding to \( R \). It follows that if we want to count the number of regions of \( R \) intersected by a query segment \( \overline{uv} \) on line \( \ell \) we have to count the number of strips in \( R^* \) containing the point \( \ell^* \) and whose slope \( \text{slope}(R) \) lies in the range \([u_x, v_x]\). See Fig. 3 for an illustration.

\textbf{Observation 4.} Let \( p^* \) be a line, oriented from left to right, and let \( R^* \) be a strip. The line \( p^* \) intersects the bottom boundary of \( R^* \) before the top boundary of \( R^* \) if and only if \( \text{slope}(p^*) > \text{slope}(R^*) \).

---

\footnote{Alternatively, we can dualize the points into lines and use the dual arrangement to obtain all radial orders in \( O(n^2) \) time.}
Consider traversing a line \( p^* \) of \( A \) (from left to right), and let \( T_{p^*}(\ell^*) \) be the number of strips that contain the point \( \ell^* \) and that we enter through the top boundary of the strip.

Lemma 5. Let \( \ell_{uv}^* \), with \( u_x < v_x \), be a vertex of \( A \). The number of strips from \( R^* \) containing \( \ell_{uv}^* \) with a slope in \([u_x, v_x]\) is \( T_{u^*}(\ell_{uv}^*) - T_{v^*}(\ell_{uv}^*) \).

Corollary 6. Let \( u, v \in \mathbb{R} \) be bottom endpoints. The number of regions of \( R \) intersected by \( uv \) is \( T_{u^*}(\ell_{uv}^*) - T_{v^*}(\ell_{uv}^*) \).

We can easily compute the counts \( T_{u^*}(\ell_{uv}^*) \) for every vertex \( \ell_{uv}^* \) on \( u^* \) by traversing the line \( u^* \). Thus, we can compute the number of regions in \( R \) intersected by \( uv \), for all bottom endpoints \( u \) and \( v \) in a total of \( O(n^2) \) time.

Together with our dynamic programming approach for computing \( k^* \) we then get:

Theorem 7. Given a set of \( n \) vertical line segments \( R \), we can compute the maximum number of regions \( k^* \) visitable by an upper convex transversal \( Q \) in \( O(n^2) \) time.

2.2 Computing a convex transversal

We now consider computing a convex transversal that maximizes the number of regions visited. We first prove some properties of an optimal convex transversal. We then use these properties to compute the maximum number of regions visitable by such a transversal using dynamic programming.

Canonical Transversals. Like in the case of the upper hull, we first argue that we can discretize the problem. Similar to Lemma 1 we can argue that the strictly convex vertices in the upper and lower hulls must lie on endpoints of the segments in \( R \). We can then show that the leftmost and rightmost vertex must lie on the intersection point of a segment with a line that goes through at least two endpoints. Next, we give a more precise characterization of the type of transversals we have to consider.

A convex transversal \( Q' \) of \( R \) is a lower canonical transversal if and only if

- the strictly convex vertices on the upper hull of \( Q' \) lie on bottom endpoints in \( R \),
- the strictly convex vertices on the lower hull of \( Q' \) lie on bottom or top endpoints of regions in \( R \),
- the leftmost vertex \( \ell \) of \( Q' \) lies on a line through \( w \), where \( w \) is the leftmost strictly convex vertex of the lower hull of \( Q' \), and another endpoint.
- the rightmost vertex \( r \) of \( Q' \) lies on a line through \( z \), where \( z \) is the rightmost strictly convex vertex of the lower hull of \( Q' \), and another endpoint.

Let \( Q = \ell uv \) be a quadrilateral whose leftmost vertex is \( \ell \), whose top vertex is \( u \), whose rightmost vertex is \( r \), and whose bottom vertex is \( v \). A quadrilateral \( Q \) is a lower canonical quadrilateral if and only if

- \( u \) and \( v \) lie on endpoints in \( \mathbb{R} \),
- \( \ell \) lies on a line through \( v \) and another endpoint, and
- \( r \) lies on a line through \( v \) and another endpoint.

We define an upper canonical transversal, and an upper canonical quadrilateral analogously. In this case the points \( \ell \) and \( r \) are defined by points on the upper hull.

Let \( k^* \) be the maximal number of regions of \( R \) visitable by an upper convex transversal, let \( k^* \) be the maximal number of regions of \( R \) visitable by a canonical upper quadrilateral, and
let $k^u$ denote the maximal number of regions of $\mathcal{R}$ visitable by a canonical upper transversal. We define $k^u_2, k^u_4$, and $k^u$, for the maximal number of regions of $\mathcal{R}$, visitable by a lower convex transversal, canonical lower quadrilateral, and canonical lower transversal, respectively.

Lemma 8. Let $k^*$ be the maximal number of regions in $\mathcal{R}$ visitable by a convex transversal of $\mathcal{R}$. We have that $k^* = \max\{k^u_2, k^u_4, k^u, k^b_2, k^b_4, k^b\}$.

By Lemma 8 we can restrict our attention to upper and lower convex transversals, canonical quadrilaterals, and canonical transversals. We can compute an optimal upper (lower) convex transversal in $O(n^2)$ time using the algorithm from the previous section. We now argue that we can compute an optimal canonical quadrilateral in $O(n^5)$ time, and an optimal canonical transversal in $O(n^6)$ time. Arkin et al. [2] describe an algorithm that given a discrete set of vertex locations can find a convex polygon (on these locations) that maximizes the number of regions stabbed. Note, however, that since a region contains multiple vertex locations – and we may use only one of them – we cannot directly apply their algorithm.

Computing the maximal number of regions intersected by a canonical quadrilateral. Consider a canonical lower quadrilateral $Q = \ell urw$ with $u_x < w_x$. We explicitly compute the regions intersected by $\overline{u\ell} \cup \overline{lw}$ and set these aside. Using a rotational sweep we then compute how many of the remaining regions intersect $\overline{ur} \cup \overline{wr}$, for all candidate points $r$, and find the candidate point $r$ that maximizes the total number of regions intersected by $Q$. If $u_x > w_x$, we use a symmetric procedure in which we count all regions intersected by $\overline{ur} \cup \overline{rw}$ first, and then the remaining regions intersected by $\overline{u\ell} \cup \overline{lw}$.

Since there are $O(n^4)$ candidate triples $u, w, \ell$, naively computing the maximum as sketched above requires $O(n^6)$ time. We argue that we do not have to do this rotational sweep for every such triple. This reduces the running time to $O(n^5)$.

Lemma 9. Given a set of $n$ vertical line segments $\mathcal{R}$, we can compute the maximum number of regions $k^*$ visitable by a canonical quadrilateral $Q$ in $O(n^5)$ time.

Computing the maximal number of regions intersected by a canonical transversal. We describe an algorithm to compute the maximal number of regions visitable by a lower canonical convex transversal. Our algorithm consists of three dynamic programming phases, in which we consider (partial) convex hulls of a particular “shape”.

In the first phase we compute (and memorize) the maximal number of regions $B[w, u, v, \ell]$ visitable by a transversal that has $\overline{u\ell}$ as a segment in the lower hull, and a convex chain $\ell, \ldots, u, v$ as upper hull. See Fig. 4(a).

In the second phase we compute the maximal number of regions $K[u, v, w, z]$ visitable by the canonical convex transversal whose rightmost top edge is $\overline{uw}$ and whose rightmost bottom edge is $\overline{uw}$. See Fig. 4(b) and (c). To make sure that we appropriately count segments that intersect both the upper and lower hull we have to distinguish between two cases, depending on whether $u_x \leq w_x$ or vice versa. Furthermore, we use that for all pairs of candidate edges $\overline{uw}$ and $\overline{uv}$ we can precompute the number of segments $I[w, z, u, v]$ intersected by $\overline{uw}$ that are not intersected by $\overline{uv}$.

In the third phase we compute the maximal number of regions visitable when we “close” the transversal using the rightmost vertex $r$. To this end, we define $R'[z, u, v, r]$ as the number of regions visitable by the canonical transversal whose rightmost upper segment is $\overline{uw}$ and whose rightmost lower segment is $\overline{uw}$ and $r$ is defined by the strictly convex vertex $z$. 

Theorem 10. Given a set of $n$ vertical line segments $R$, we can compute the maximum number of regions $k^*$ visitable by a convex transversal $Q$ in $O(n^6)$ time.

3 2-oriented disjoint line segments

In this section we consider the case when $R$ consists of vertical and horizontal disjoint segments. We will show how to apply similar ideas to those presented in the previous section to compute an optimal convex transversal $Q$ of $R$. As in the previous section, we will mostly restrict our search to canonical transversals. However, we will have one special case to consider when an optimal partial convex transversal has bends not necessarily belonging to a discrete set of points. In this section we will provide an overview of the ideas behind our approach; the reader is referred to the full version of this paper for the missing details [9].

We call the left-, right-, top- and bottommost vertices $\ell$, $r$, $u$ and $b$ of $Q$ the extreme vertices. They subdivide the transversal into four chains. Similarly to the 1-oriented case, we can move the non-extreme convex vertices to be on the endpoints of the segments (Lemma 11). In the 1-oriented case, the extreme vertices were restricted to being on intersections of lines through endpoints with segments in $R$, which we will call a 1st-order fixed point. For the 2-oriented case, we need to extend this notion: when one extreme vertex is on a 1st-order fixed point, the opposite extreme vertex might be on the intersection of a line through an endpoint and a 1st-order fixed point, the opposite extreme vertex might be on the intersection of a line through an endpoint and the 1st-order fixed point with a segment in $R$ (these are 2nd-order fixed points). The proof is analogous to that of Lemma 1.

Lemma 11. Let $Q$ be a convex partial transversal of $R$ with extreme vertices $\ell$, $r$, $t$, and $b$. There exists a convex partial transversal $Q'$ of $R$ such that

- the two transversals have the same extreme vertices,
- all segments that are intersected by the upper-left, upper-right, lower-right, and lower-left hulls of $Q$ are also intersected by the corresponding hulls of $Q'$,
- all strictly convex vertices on the upper-left hull of $Q'$ lie on bottom endpoints of vertical segments or on the right endpoints of horizontal segments of $R$, and
- the convex vertices on the other hulls of $Q'$ lie on analogous endpoints.
Let $Q$ be the maximum convex transversal. There are three cases to consider. (1) There exists a chain of the convex hull of $Q$ containing at least two endpoints of segments, (2) there exists a chain of the convex hull of $Q$ containing no endpoints, or (3) all four convex chains contain at most one endpoint. In case (1) we prove that one can move the endpoints around such that all points of the transversal are on a discrete set of points, allowing us to search for a canonical transversal (see below). In case (2) one can move the extreme points adjacent to that chain in such a way that the chain encounters an endpoint. In case (3) we can either move the points around such that one chain now contains two endpoints, putting us in case (1), or we are in the “special case” that is solved separately.

### 3.1 Calculating the canonical transversal

We subdivide our problem into subproblems (shown in Figure 5(a)) that can be solved using the algorithm for the 1-oriented case. We observe that if we fix the extreme vertices, we have a partial ordering of segments on each chain, defining the order in which they can be intersected. For each chain, we guess a point that will be a vertex. This gives us a subproblem for each extreme point: we need to find an “upper” and “lower” chain that links the extreme point to the guessed vertices. For this we can simply use the algorithm for the parallel case, except in the case where there are segments in $R$ that could intersect two non-adjacent chains. We put such segments into a separate subproblem, of which there can be only one. We then need to examine all possible combinations of extreme points and guessed vertices, but as this is a constant number of points, and as we choose them out of a polynomial number of points, this gives a polynomial time algorithm. This algorithm extends to any constant number of orientations.

### 3.2 Special case

As mentioned above this case only occurs when the four hulls each contain exactly one endpoint. The construction can be seen in Figure 5(b). Let $e_{ud}$, $e_{ur}$, $e_{br}$ and $e_{bd}$ be the endpoints on the upper-left, upper-right, lower-right and lower-left hull. Let further $s_u$, $s_r$, $s_b$ and $s_l$ be the segments that contain the extreme points.

For two points $a$ and $b$, let $l(a, b)$ be the line through $a$ and $b$. For a given position of $u$ we can place $r$ on or below the line $l(u, e_{ur})$. Then we can place $b$ on or left of the line $l(r, e_{br})$, $l$ on or above $l(b, e_{bd})$ and then test if $u$ is on or to the right of $l(l, e_{ul})$. Placing $r$ lower decreases the area where $b$ can be placed and the same holds for the other extreme points.
It follows that we place \( r \) on the intersection of \( l(u, e_{ur}) \) and \( s_r \), we set \( \{b\} = l(r, e_{br}) \cap s_b \) and \( \{\ell\} = l(b, e_{\ell\theta}) \cap s_{\ell\theta} \). Let then \( u' \) be the intersection of the line \( l(\ell, e_{\ell\theta}) \) and the upper segment \( s_{u\tau} \). In order to make the test if \( u' \) is left of \( u \) we first need the following lemma.

Lemma 12. Given a line \( \ell \), a point \( A \), and a point \( X(\tau) \) with coordinates \( \left( \frac{P_2(\tau)}{Q(\tau)}, \frac{P_1(\tau)}{Q(\tau)} \right) \) where \( P_1(\cdot), P_2(\cdot), \) and \( Q(\cdot) \) are linear functions. The intersection \( Y \) of \( \ell \) and the line through the points \( X \) and \( A \) has coordinates \( \left( \frac{P'_2(\tau)}{Q'(\tau)}, \frac{P'_1(\tau)}{Q'(\tau)} \right) \) where \( P'_1(\cdot), P'_2(\cdot) \) and \( Q'(\cdot) \) are linear functions.

Let \((\tau, c)\) be the coordinates of the point \( u \) for \( \tau \in I \), where the constant \( c \) and the interval \( I \) are determined by the segment \( s_u \). Then by Lemma 12 we have that the points \( r, b, \ell, u' \) all have coordinates of the form specified in the lemma. First we have to check for which values of \( \tau \) the point \( u \) is between \( e_{ur} \) and \( e_{ur} \), \( r \) is between \( e_{br} \) and \( e_{ur} \), \( b \) is between \( e_{\ell\theta} \) and \( e_{ur} \) and \( \ell \) is between \( e_{\ell\theta} \) and \( e_{rt} \). This results in a system of linear equations whose solution is an interval \( I' \).

We then determine the values of \( \tau \in I' \) where \( u' = \left( \frac{P_2(\tau)}{Q(\tau)}, \frac{P_1(\tau)}{Q(\tau)} \right) \) is left of \( u = (\tau, c) \) by considering the following quadratic inequality: \( \frac{P_1(\tau)}{Q(\tau)} \leq \tau \). If there exists a \( \tau \) satisfying all these constraints, then there exists a convex transversal such that the points \( u, r, b, \ell \) are the top-, right-, bottom-, and leftmost points, and the points \( e_{jk} \) \((j, k = u, r, b, \ell)\) are the only endpoints contained in the hulls.

Combining this with the algorithm in the previous section, we obtain the following result:

Theorem 13. Given a set of 2-oriented line segments, we can compute the maximum number of regions visited by a convex partial transversal in polynomial time.

Extensions. One should note that the concepts explained here generalize to more orientations. For each additional orientation there will be two more extreme points and therefore two more chains. It follows that for \( \rho \) orientations there might be \( \rho \)-th-order fixed points. This increases the running time, because more points need to be guessed and the pool of discrete points to choose from is bigger, but for a fixed number of orientations it is still polynomial in \( n \). The special case generalizes as well, which means that the same case distinction can be used. We further know that when \( R \) is a set of non-intersecting connected regions, any transversal with size at least 2 intersects the boundary of each region containing a point of the transversal. It follows that the algorithm extends to disjoint convex polygons with limited edge orientations, e.g. disjoint axis-aligned rectangles.

4 3-oriented intersecting segments

We prove that the problem of finding a maximum convex partial transversal \( Q \) of a set of 3-oriented segments \( R \) is NP-hard using a reduction from Max-2-SAT.

Theorem 14. Let \( R \) be a set of segments that have three different orientations. The problem of finding a maximum convex partial transversal \( Q \) of \( R \) is NP-hard.

First, note that we can choose the three orientations without loss of generality: any (non-degenerate) set of three orientations can be mapped to any other set using an affine transformation, which preserves convexity of transversals. We choose the three orientations in our construction to be vertical (\( \| \)), the slope of 1 (\( / \)) and the slope of -1 (\( \backslash \)).

Given an instance of MAX-2-SAT we construct a set of segments \( R \) and then we prove that from a maximum convex partial transversal \( Q \) of \( R \) one can deduce the maximum number of clauses that can be made true in the instance.
Figure 6 Overview of our construction. Each of the colored segment chains represents a variable. At each point where a chain bounces on the banana there is a fruit fly gadget. At each area marked orange there is a clause gadget. Each chain is only pictured once, but in actuality each chain is copied $m + 1$ times and placed at distance $\epsilon$ of each other. The distance between the different variables is exaggerated for clarity.

Figure 7 Sketch of a fly gadget. Endpoints of chain segments are divided over two implicit parabolic arcs together with some extra regions. To maximize our transversal, one of the two implicit arcs must be picked. This choice propagates through the rest of the construction. In our actual construction, the fly appears completely swatted: the aspect ratio of the fly approaches the local curvature of the banana, making it almost completely flat. The outer chain segments are then at an angle of 90°.

4.1 Overview of the construction

Our constructed set $\mathcal{R}$ consists of several different substructures. The construction is built inside a long and thin rectangle, referred to as the crate. The crate is not explicitly part of $\mathcal{R}$. Inside the crate, for each variable, there are several sets of segments that form chains. These chains alternate $\backslash$ and $/$ segments reflecting on the boundary of the crate. The idea is that an optimal solution must always place a point at (or close to) one of the endpoints of these segments, and furthermore, that two adjacent segments cannot both have their point at the reflection point. For each clause, there are vertical $|$ segments to transfer the state of a variable to the opposite side of the crate. Figure 6 shows this idea. However, the segments do not extend all the way to the boundary of the crate; instead they end on the boundary of a slightly smaller convex shape inside the crate, which we call the banana. By having all of the endpoints on the banana, the maximum partial transversal will be strictly convex. Aside from the chains associated with variables, $\mathcal{R}$ also contains segments that form gadgets to ensure that the variable chains have a consistent state, and gadgets to represent the clauses of our MAX-2-SAT instance. Due to their winged shape, we refer to these gadgets by the name fruit flies. The idea is that an optimal solution must use one of two sequences of small points above the wings of the flies, and depending on this choice, can use only the endpoints of segments ending in one of the wings of the fly. See Figure 7 for a sketch of a fruit fly.
Our construction makes it so that we can always find a transversal that includes all of
the chains, the maximum number of segments on the gadgets, and half of the | segments.
For each clause of our MAX-2-SAT instance that can be satisfied, we can also include one
of the remaining | segments. For the full construction and proof of correctness, see the full
version of this paper [9].

Implications. Our construction strengthens the proof in [13] by showing that using only 3
orientations, the problem is already NP-hard. The machinery appears to be powerful: with a
slight adaptation, we can also show that the problem is NP-hard for axis-aligned rectangles.

Theorem 15. Let \( \mathcal{R} \) be a set of (potentially intersecting) axis-aligned rectangles. The
problem of finding a maximum convex partial transversal \( Q \) of \( \mathcal{R} \) is NP-hard.

Proof. We build exactly the same construction, but afterwards we replace every vertical
segment by a \( 45^\circ \) rotated square and all other segments by arbitrarily thin rectangles. The
points on the banana’s boundary are opposite corners of the square, and the body of the
square lies in the interior of the banana so placing points there is not helpful.

References

   Conflict-free Covering. In Proc. 27th Canadian Conference on Computational Geometry (CCCG),
2. E. M. Arkin, C. Dieckmann, C. Knauer, J. S. B. Mitchell, V. Polishchuk, L. Schlipf, and
3. M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. Computational Geometry:
   regions with applications to frequency assignment in cellular networks. SIAM Journal on
7. S. Har-Peled and S. Smorodinsky. Conflict-free coloring of points and simple regions in the
9. V. Keikha, M. van de Kerkhof, M. van Kreveld, I. Kostitsyna, M. Löffler, F. Staals,
   J. Urhausen, J. L. Vermeulen, and L. Wiratma. Convex partial transversals of planar
    0020-0190(95)00130-5.
12. G. Rote, G. Woeginger, B. Zhu, and Z. Wang. Counting \( k \)-subsets and convex \( k \)-gons in
Extending the Centerpoint Theorem to Multiple Points

Alexander Pilz
Institute of Software Technology, Graz University of Technology, Austria
apilz@ist.tugraz.at
https://orcid.org/0000-0002-6059-1821

Patrick Schnider
Department of Computer Science, ETH Zurich, Switzerland
patrick.schnider@inf.ethz.ch

Abstract
The centerpoint theorem is a well-known and widely used result in discrete geometry. It states that for any point set $P$ of $n$ points in $\mathbb{R}^d$, there is a point $c$, not necessarily from $P$, such that each halfspace containing $c$ contains at least $\frac{n}{d+1}$ points of $P$. Such a point $c$ is called a centerpoint, and it can be viewed as a generalization of a median to higher dimensions. In other words, a centerpoint can be interpreted as a good representative for the point set $P$. But what if we allow more than one representative? For example in one-dimensional data sets, often certain quantiles are chosen as representatives instead of the median.

We present a possible extension of the concept of quantiles to higher dimensions. The idea is to find a set $Q$ of (few) points such that every halfspace that contains one point of $Q$ contains a large fraction of the points of $P$ and every halfspace that contains more of $Q$ contains an even larger fraction of $P$. This setting is comparable to the well-studied concepts of weak $\varepsilon$-nets and weak $\varepsilon$-approximations, where it is stronger than the former but weaker than the latter. We show that for any point set of size $n$ in $\mathbb{R}^d$ and for any positive $\alpha_1, \ldots, \alpha_k$ where $\alpha_1 \leq \alpha_2 \leq \ldots \leq \alpha_k$ and for every $i, j$ with $i + j \leq k + 1$ we have that $(d-1)\alpha_k + \alpha_i + \alpha_j \leq 1$, we can find $Q$ of size $k$ such that each halfspace containing $j$ points of $Q$ contains least $\alpha_jn$ points of $P$. For two-dimensional point sets we further show that for every $\alpha$ and $\beta$ with $\alpha \leq \beta$ and $\alpha + \beta \leq \frac{2}{3}$ we can find $Q$ with $|Q| = 3$ such that each halfplane containing one point of $Q$ contains at least $\alpha n$ of the points of $P$ and each halfplane containing all of $Q$ contains at least $\beta n$ points of $P$. All these results generalize to the setting where $P$ is any mass distribution. For the case where $P$ is a point set in $\mathbb{R}^2$ and $|Q| = 2$, we provide algorithms to find such points in time $O(n \log^2 n)$.

2012 ACM Subject Classification Design and analysis of algorithms

Keywords and phrases centerpoint, point sets, Tukey depth

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.53


Acknowledgements We thank Emo Welzl for initiating discussions on this topic, as well as anonymous reviewers for helpful comments.

1 Supported by a Schrödinger fellowship of the Austrian Science Fund (FWF): J-3847-N35.
1 Introduction

Medians and quantiles are ubiquitous in the statistical analysis and visualization of data. These notions allow for quantifying how deep some point lies within a one-dimensional data set by measuring how many elements of the data set appear before the point and how many appear after it. In comparison to the mean, medians and quantiles have the advantage that they only depend on the order of the data points, and not their exact positions, making them robust against outliers. However, in many applications, data sets are multidimensional, and there is no clear order of the data set. For this reason, various generalizations of medians to higher dimensions have been introduced and studied. Many of these generalized medians rely on a notion of depth of a query point within a data set, a median then being a query point with the highest depth among all possible query points. Several such depth measures have been introduced over time, most famously Tukey depth [18] (also called halfspace depth), simplicial depth, or convex hull peeling depth (see, e.g., [1]). All of these depth measures lead to generalized medians that are invariant under affine transformations. As for quantiles, only a few generalizations have been introduced (see, e.g., [6]). We propose such a generalization by extending a depth measure to sets with a fixed number of query points and defining a quantile as a set with maximal depth. The depth measure we extend is Tukey depth: the Tukey depth of a point \( q \) with respect to a point set \( P \subseteq \mathbb{R}^d \) is the minimal number of points of \( P \) in any closed halfspace containing \( q \). More formally, if \( H \) denotes the set of closed halfspaces, then the Tukey depth \( \text{td}_P(q) \) of \( q \) with respect to \( P \) is

\[
\text{td}_P(q) = \min_{h \in H} \{|h \cap P|\}.
\]

Similarly, the Tukey depth can also be defined for a mass distribution \( \mu \):

\[
\text{td}_\mu(q) = \min_{h \in H} \{\mu(h)\}.
\]

Here, a mass distribution \( \mu \) on \( \mathbb{R}^d \) is a measure on \( \mathbb{R}^d \) such that all open subsets of \( \mathbb{R}^d \) are measurable, \( 0 < \mu(\mathbb{R}^d) < \infty \) and \( \mu(S) = 0 \) for every lower-dimensional subset \( S \) of \( \mathbb{R}^d \).

The centerpoint theorem states that there is always a point of high depth, i.e., a point \( q \) such that for every closed halfspace \( h \) containing \( q \) we have \( |h \cap P| \geq \frac{|P|}{d+1} \) (or \( \mu(h) \geq \frac{\mu(\mathbb{R}^d)}{d+1} \)) for masses. Note that, for \( d = 1 \), such a centerpoint is a median: a median has the property that every halfline containing it contains at least half of the underlying data set. Quantiles can be interpreted similarly: the \( \frac{1}{3} \)-quantile and the \( \frac{2}{3} \)-quantile form a set of two points such that every halfline that contains one of them contains at least \( \frac{1}{3} \) of the data set. Furthermore, a halfline containing both of the points contains at least \( \frac{2}{3} \) of the underlying data set. In particular, halflines containing more points contain more of the data set. This idea leads to the following generalization of Tukey depth for a set \( Q \) of multiple points:

\[
\text{gtd}_P(Q) := \min_{h \in H : Q \cap h \neq \emptyset} \left\{ \frac{|h \cap P|}{|h \cap Q|} \right\}.
\]

Again, we can generalize this to mass distributions:

\[
\text{gtd}_\mu(Q) := \min_{h \in H : Q \cap h \neq \emptyset} \left\{ \frac{\mu(h)}{|h \cap Q|} \right\}.
\]

We prove that there is always a set \( Q \) of \( k \) points that has generalized Tukey depth \( \frac{1}{d+1} \). In fact, we prove the following, more general statement:
Theorem 1. Let $\mu$ be a mass distribution in $\mathbb{R}^d$ with $\mu(\mathbb{R}^d) = 1$. Let $\alpha_1, \ldots, \alpha_k$ be non-negative real numbers such that $\alpha_1 \leq \alpha_2 \leq \ldots \leq \alpha_k$ and for every $i,j$ with $i+j \leq k+1$ we have that $(d-1)\alpha_i + \alpha_j \leq 1$. Then there are $k$ points $p_1, \ldots, p_k$ in $\mathbb{R}^d$ such that for each closed halfspace $h$ containing $j$ of the points $p_1, \ldots, p_k$ we have $\mu(h) \geq \alpha_j$.

Note that, for $d=1$ and $k=2$, the points $p_1$ and $p_2$ correspond to the $\alpha_1$-quantile and the $(1-\alpha_1)$-quantile; for $\alpha_j = \frac{j}{d+1}$ we get our bound on the generalized Tukey depth, and for $\alpha_1 = \ldots = \alpha_k$, the result implies the centerpoint theorem.

Our second result is motivated by interpreting the $\frac{1}{d}$-quantile and the $\frac{2}{d}$-quantile not as two points, but as a one-dimensional simplex. We then have that every halfline that contains a part of the simplex contains at least $\frac{1}{d}$ of the underlying data set and every halfline that contains the whole simplex contains at least $\frac{2}{d}$ of the underlying data set. Also for this interpretation we give a generalization to two dimensions:

Theorem 2. Let $\mu$ be a mass distribution in $\mathbb{R}^2$ with $\mu(\mathbb{R}^2) = 1$. Let $\alpha$ and $\beta$ be real numbers such that $0 < \alpha \leq \beta$ and $\alpha + \beta = \frac{2}{d}$. Then there is a triangle $\Delta$ in $\mathbb{R}^2$ such that

1. for each closed halfplane $h$ containing one of the vertices of $\Delta$ we have $\mu(h) \geq \alpha$ and
2. for each closed halfplane $h$ fully containing $\Delta$ we have $\mu(h) \geq \beta$.

Note that this again generalizes centerpoints for $\alpha = \beta$. However, this result does not give bounds on the generalized Tukey depth of these sets, as, e.g., a halfspace containing two points may still only contain an $\alpha$-fraction of the mass.

Finally, we give algorithms to compute two points satisfying the two-dimensional case of Theorem 1 and three points satisfying Theorem 2 in time $O(n \log^2 n)$.

Related work. Another way to view our setting is the following: given a multidimensional data set, we want to find a fixed number of representatives. The idea of small point sets representing a larger point set has been studied in many different settings. One of the most famous of those is the concept of $\varepsilon$-nets, introduced by Haussler and Welzl [7]. For a range space $(X, R)$, consisting of a set $X$ and a set $R$ of subsets of $X$, an $\varepsilon$-net on $P \subset X$ is a subset $N$ of $P$ with the property that every $r \in R$ with $|r \cap P| \geq \varepsilon|P|$ intersects $N$. In our setting, where we consider halfspaces, we would choose $X = \mathbb{R}^d$ and $R$ as the set of all halfspaces. It is known that for this range space, for any point set $P$ there exists an $\varepsilon$-net of size $O(\frac{\varepsilon}{\varepsilon} \log \frac{1}{\varepsilon})$. In particular, this bound does not depend on the size of $P$. Note that we require the $\varepsilon$-net to be a subset of $P$. If this condition is dropped, we arrive at the concept of weak $\varepsilon$-nets. The fact that the points for the weak $\varepsilon$-net can be chosen anywhere in $\mathbb{R}^d$ allows for very small weak $\varepsilon$-nets for many range spaces. There has been some work on weak $\varepsilon$-nets of small size. For halfplanes in $\mathbb{R}^2$ for example, Aronov et al. [3] have shown that there is always a weak $\frac{1}{2}$-net of two points. These two points both lie outside of the convex hull of $P$. They also consider many other range spaces, such as convex sets, disks and rectangles. Similarly, Babazadeh and Zarrabi-Zadeh [4] construct weak $\frac{1}{2}$-nets of size 3 for halfspaces in $\mathbb{R}^3$. For two-dimensional convex sets, Mustafa and Ray [15] have shown that there is always a weak $\frac{2}{3}$-net of two points; Shabbir [17] shows how to find two such points in $O(n \log^4 n)$ time.

Another related concept is the concept of $\varepsilon$-approximations: For a range space $(X, R)$ an $\varepsilon$-approximation on $P \subset X$ is a subset $N$ of $P$ with the property that for every $r \in R$ we have $\frac{|r \cap N|}{|r \cap P|} \leq \varepsilon$. Again, the restriction that $N$ has to be a subset of $P$ can be dropped to get the notion of weak $\varepsilon$-approximations. Just as for $\varepsilon$-nets, there has been a lot of work on $\varepsilon$-approximations and weak $\varepsilon$-approximations, see [14] for a recent survey. In particular it was shown that for halfspaces in $\mathbb{R}^d$, there always is an $\varepsilon$-approximation of size $O(\frac{1}{\varepsilon^2 (\varepsilon + 1)})$ [12, 13].
While our setting can be considered to be related to weak $\varepsilon$-nets and weak $\varepsilon$-approximations for range spaces determined by halfspaces, the differences are significant. As we will discuss here, a halfspace missing all the points of $Q$ may still contain up to half of the points of the initial set, and thus $Q$ qualifies neither as a good weak $\varepsilon$-approximation nor $\varepsilon$-net.

Note that for $|Q| = 2$, the condition of Theorem 1 that any halfspace containing all of the points of $Q$ contains at least $\alpha_2 n$ points of $P$ is equivalent to the statement that every halfspace containing more than $(1 - \alpha_2) n$ of the points of $P$ contains at least one point of $Q$. So, $Q$ is a weak $(1 - \alpha_2)$-net of $P$. Furthermore, the condition that any halfspace containing one of the points of $Q$ contains at least $\alpha_1 n$ points of $P$ translates to the statement that every halfspace containing more than $(1 - \alpha_1) n$ of the points of $P$ must contain all of $Q$. Thus, $Q$ is not only a weak $(1 - \alpha_2)$-net of $P$ but also has the additional property that all points of $Q$ are somewhat deep within $P$. (For two points in the plane, this comes at the cost of having $\varepsilon$ larger than $\frac{1}{2}$.) On the other hand, while we require halfspaces containing all points of $Q$ to also contain many points of $P$, we also allow halfspaces containing only one point of $Q$ to contain many points of $P$. This separates our concept from weak $\varepsilon$-approximations. Note that when dealing with halfspaces and $\varepsilon$-nets of size 2, the weak $\frac{1}{2}$-net of Aronov et al. [3] is actually also a weak $\frac{1}{2}$-approximation. Similarly, Theorem 1 gives us a weak $(1 - \alpha_2)$-approximation of $P$, with the optimal value being reached when $\alpha_1$ tends to zero (which actually corresponds to the result in [3]). Adding more points to $Q$ does not give us a better approximation. For $d = 2$, requiring that for $i + j \leq k + 1$ we have $(d - 1)\alpha_i + \alpha_j < 1$ implies $\alpha_1 + 2\alpha_k < 1$, so a halfspace containing no points of $Q$ may contain half of the points of $P$; we therefore cannot get anything better than a weak $\frac{1}{2}$-approximation. Also, we do not get anything better than a weak $\frac{1}{2}$-net.

In fact, our setting is very much related to the concept of one-sided $\varepsilon$-approximants, recently introduced by Bukh and Nivasch [5]: For a range space $(X, R)$, a one-sided $\varepsilon$-approximant on $P \subset X$ is a subset $N$ of $P$ with the property that for every $r \in R$ we have $|n^r|P| - |n^r|N| \leq \varepsilon$. Once again, the restriction that $N$ has to be a subset of $P$ can be dropped to get the notion of weak one-sided $\varepsilon$-approximations. Note that the only difference to the definition of $\varepsilon$-approximations is that one does not take the absolute value of the difference. In particular, if $|n^r|N| > |n^r|P|$, i.e., if $r$ contains many points of $N$ despite containing only few points of $P$, the difference is negative, and thus smaller than $\varepsilon$.

In their paper, Bukh and Nivasch [5] consider the range space of convex sets. They show that any point set in $\mathbb{R}^d$ allows a one-sided $\varepsilon$-approximant for convex ranges of size $g(\varepsilon, d)$, where $g(\varepsilon, d)$ only depends on $\varepsilon$ and $d$, but not on the size of $P$.

In a similar reasoning, it makes sense to define an approximation by a set $N$ such that for every $r \in R$ we have $|n^r|N|/|N| - |n^r|P|/|P| \leq \varepsilon$. Intuitively, if a range $r$ contains a large fraction of the points of $N$, then it is guaranteed to contain a large fraction of the set $P$ we want to approximate. But here again, our approximation ratio is $\frac{1}{2}$ at best.

## 2 Two points

We first consider the case where the underlying data is a point set. Motivated by the definition of generalized Tukey depth, we consider $\alpha_1 = \frac{1}{6}$ and $\alpha_2 = \frac{2}{5}$. Even though this result is a special case of Theorem 1, we still show its proof for two reasons: first, the Algorithm presented in Section 5 relies heavily on the presented proof and, secondly, the proof already illustrates the main ideas for the proof of Theorem 1.
Theorem 3. Let $P$ be a set of $n$ points in general position in the plane. Then there are two points $p_1$ and $p_2$ in $\mathbb{R}^2$ such that

1. each closed halfplane containing one of the points $p_1$ and $p_2$ contains at least $\frac{n}{2}$ of the points of $P$ and
2. each closed halfplane containing both $p_1$ and $p_2$ contains at least $\frac{2n}{5}$ of the points of $P$.

Proof. Note that condition (1) is equivalent to the condition that every open halfplane containing more than $\frac{4n}{5}$ of the points of $P$ must contain both $p_1$ and $p_2$. Similarly, condition (2) is equivalent to the condition that every open halfplane containing more than $\frac{3n}{5}$ of the points of $P$ must contain one of $p_1$ and $p_2$. We will now construct two points $p_1$ and $p_2$ satisfying both these conditions.

Let $C$ be the intersection of all open halfplanes containing more than $\frac{4n}{5}$ of the points of $P$. Clearly $C$ is convex. Also, note that $C$ is closed. The centerpoint region is a strict subset of $C$ and thus $C$ has a non-empty interior. In order to satisfy condition (1), both $p_1$ and $p_2$ must contain $C$.

Let now $H$ be the set of all open halfplanes containing more than $\frac{2n}{5}$ of the points of $P$. For any $h_1$ in $H$ let $c_1$ be the intersection of $h_1$ and $C$. In order to also satisfy condition (2), we need to find two points $p_1$ and $p_2$ such that every $c_1$ contains at least one of them. To this end, we partition $H$ into two subsets $L$ and $R$. The set $L$ contains all halfplanes that lie on the left side of their respective boundary lines. Analogously, $R$ contains all halfplanes that lie on the right side of their respective boundary lines. For a halfplane $h$, that has a horizontal boundary line, we put $h_1$ in $L$ if and only if it lies above its boundary line.

Note that any three halfplanes in $L$ have a non-empty intersection: Consider the inclusion-minimal halfplane $h \in L$ with horizontal boundary line and its intersection $r$ with the boundary of the convex hull of $P$. As $h$ is open, $r$ is not in $h$. However, we claim that any point $r'$ in $h$ on the convex hull boundary of $P$ in an $\varepsilon$-neighborhood of $r$ is in any halfplane of $L$. Indeed, if there was a halfplane in $L$ not containing $r'$, it would contain a strict subset of the intersection of the convex hull of $P$ with $h$; however, this would contradict the minimality of $h$. The analogous holds for $R$.

We will now show that for any two halfplanes $h_1$ and $h_2$ in $L$, their corresponding regions $c_1$ and $c_2$ have a non-empty intersection. The same arguments hold for any two halfplanes in $R$. Assume for the sake of contradiction that $c_1$ and $c_2$ do not intersect. As $C$ and $h_1 \cap h_2$ are convex, this means that there is an open halfplane $g$ containing more than $\frac{4n}{5}$ of the points of $P$ such that the intersection of the boundary lines of $h_1$ and $h_2$ lies in $g$, the complement of $h$ (see Figure 1). In particular, $g \cap h_1$ is a strict subset of $h_2$. As $\overline{g}$ contains strictly fewer than $\frac{n}{2}$ of the points of $P$ and $\overline{h}_1$ contains strictly fewer than $\frac{2n}{5}$ of the points of $P$, $g \cap h_1$ must contain strictly more than $\frac{2n}{5}$ of the points of $P$. However, being a subset of $\overline{h}_2$, which also contains strictly fewer than $\frac{2n}{5}$ of the points of $P$, this is a contradiction. Thus, by contradiction, $c_1$ and $c_2$ intersect.

As neither three halfplanes in $L$ nor two halfplanes in $L$ and $C$ have an empty intersection, Helly’s Theorem entails that there exists a point in both $C$ and all halfplanes in $L$, i.e., all $c_i$s associated to $L$ have a non-empty intersection $D_L$. Again, the same holds for $R$, with a non-empty intersection $D_R$. Placing $p_1$ in $D_L$ and $p_2$ in $D_R$, we have thus constructed two points such that the conditions (1) and (2) hold.

This result is tight in the following sense: There is a point set for which it is not possible to improve both conditions at the same time, that is, it is not possible to find two points such that any halfplane containing one of them contains strictly more than $\frac{2}{3}$ of the points and any halfplane containing both of them contains strictly more than $\frac{2n}{5}$ of the points. For
Extending the Centerpoint Theorem to Multiple Points

**Figure 1** Two circles associated to $L$ must intersect (left). The intersection is non-empty in other variants (right).

**Figure 2** A construction for which the bounds of Theorem 3 cannot be improved.

this consider a set of $n = 5k$ point arranged in the following way. Partition the points into 5 sets $P_1, \ldots, P_5$ of $k$ points each. Place $P_1, \ldots, P_5$ in such a way that the convex hull of each $P_i$ is disjoint from the convex hull of the union of the other four sets (see Figure 2).

Denote by $S_{i,j}$ the convex hull $CH(P_i \cup P_j)$ of $P_i \cup P_j$. Let $\ell$ be a line through $CH(P_i)$ and $CH(P_j)$. Note that any other set $P_m$ is not separated by $\ell$ (i.e., lies entirely on one side). Let $A_{i,j}$ be the side of $\ell$ containing fewer of the other sets and let $B_{i,j}$ be the other side. For any point $q$ in $CH(P_1 \cup \ldots \cup P_5)$ we say that $q$ is above $S_{i,j}$ if it is not in $S_{i,j}$ but it is in $A_{i,j}$. Similarly, for any point $q$ in $CH(P_1 \cup \ldots \cup P_5)$ we say that $q$ is below $S_{i,j}$ if it is not in $S_{i,j}$ but it is in $B_{i,j}$. Suppose, for the sake of contradiction, that there exist two points $p_1$ and $p_2$ such that any halfplane containing one of them contains strictly more than $k$ of the points of $P_1 \cup \ldots \cup P_5$ and any halfplane containing both of them contains strictly more than $2k$ of the points of $P_1 \cup \ldots \cup P_5$. Consider two sets $P_i$ and $P_j$ such that $A_{i,j}$ contains exactly one other set. First we note that neither $p_1$ nor $p_2$ can lie above $S_{i,j}$ as otherwise we can find a halfplane containing that point and only one of the sets, i.e., only $k$ points. Similarly, we cannot place both $p_1$ and $p_2$ below $S_{i,j}$, as otherwise we can find a halfplane containing both points and only two of the sets, i.e., only $2k$ points. Also, we must clearly place both $p_1$ and $p_2$ in $CH(P_1 \cup \ldots \cup P_5)$. Thus, for any two sets $P_i$ and $P_j$ such that $A_{i,j}$ contains exactly one other set, $S_{i,j}$ must contain at least one of $p_1$ and $p_2$. However, there are five such $S_{i,j}$ and $P_1, \ldots, P_5$ can be placed in such a way that no three of them have a common intersection. So no matter how we place $p_1$ and $p_2$, one of the $S_{i,j}$ will be empty.

### 3 An arbitrary number of points

We now strengthen Theorem 3 in four ways: we allow for arbitrarily many query points, we extend it to higher dimensions, we consider mass distributions instead of point sets, and we give a range of possible bounds:
Theorem 1. Let \( \mu \) be a mass distribution in \( \mathbb{R}^d \) with \( \mu(\mathbb{R}^d) = 1 \). Let \( \alpha_1, \ldots, \alpha_k \) be non-negative real numbers such that \( \alpha_1 \leq \alpha_2 \leq \ldots \leq \alpha_k \) and for every \( i, j \) with \( i + j \leq k + 1 \) we have that \( (d - 1)\alpha_i + \alpha_i + \alpha_j \leq 1 \). Then there are \( k \) points \( p_1, \ldots, p_k \) in \( \mathbb{R}^d \) such that for each closed halfspace \( h \) containing \( j \) of the points \( p_1, \ldots, p_k \) we have \( \mu(h) \geq \alpha_j \).

We use the following observation, which follows from the fact that for an empty intersection of \( d + 1 \) halfspaces, any point with non-zero mass is in at most \( d \) such halfspaces.

Observation 4. Let \( \mu \) be a mass distribution in \( \mathbb{R}^d \) with \( \mu(\mathbb{R}^d) = 1 \). Let \( h_1, \ldots, h_{d+1} \) be \( d + 1 \) open halfspaces with \( h_1 \cap \ldots \cap h_{d+1} = \emptyset \). Then \( \mu(h_1) + \ldots + \mu(h_{d+1}) \leq d \).

Proof of Theorem 1. The result is straightforward for \( d = 1 \), so assume \( d \geq 2 \). Again the condition that for each closed halfspace \( h' \) containing \( j \) of the points \( p_1, \ldots, p_k \) we have \( \mu(h') \geq \alpha_j \) is equivalent to the condition that every open halfspace \( h \) with \( \mu(h) > 1 - \alpha_j \) must contain at least \( k - j \) of the points \( p_1, \ldots, p_k \). Let \( \alpha_0 = 0 \). For \( 1 \leq j \leq k \), we call an open halfspace \( h \) a \( j \)-halfspace if \( 1 - \alpha_{k-j+1} < \mu(h) \leq 1 - \alpha_{k-j} \). Consider the \( x_1 \times x_2 \)-plane, denoted by \( \mathcal{X} \), and for each vector \( v = (v_1, v_2, \ldots, v_d) \) in \( \mathbb{R}^d \) let \( \pi(v) = (v_1, v_2, 0, \ldots, 0) \) be the projection of \( v \) to \( \mathcal{X} \). Let \( v_1, \ldots, v_k \) be \( k \) unit vectors in \( \mathcal{X} \) with the property that the angle between any \( v_i \) and \( v_{i+1} \) is \( \frac{2\pi}{k} \). Note that this implies that also the angle between \( v_k \) and \( v_1 \) is \( \frac{2\pi}{k} \). For each \( v_i \) we construct a principal set \( V_i \) of halfspaces as follows: For each \( j, i \) consider all \( j \)-halfspaces. For any such halfspace \( h \), let \( n(h) \) be the normal vector to its bounding hyperplane that points into \( h \). Let \( h \) be in \( V_i \) if the angle between \( \pi(n(h)) \) and \( v_i \) is at most \( \frac{2\pi}{k} \). If \( \pi(n(h)) = 0 \), place \( h \) arbitrarily in \( j \) of the \( V_i \)'s. Note that with this construction each \( j \)-halfspace is contained in exactly \( j \) principal sets. Thus, if, for each principal set, we can pick a point in all its halfplanes, then each \( j \)-halfspace contains \( j \) points.

It remains to show that the halfspaces in each principal set have a common intersection. Let \( h_1, \ldots, h_{d+1} \) be \( d + 1 \) halfspaces in \( V_i \) and assume for the sake of contradiction that they have no common intersection. Then the positive hull (conical hull) of their projected normal vectors must be \( \mathcal{X} \), and in particular there are three of them, w.l.o.g. \( h_1 \), \( h_2 \) and \( h_3 \), whose projected normal vectors already have \( \mathcal{X} \) as their positive hull. Further, among those three halfspaces, there are two of them, w.l.o.g. \( h_1 \) and \( h_2 \), such that the angles between their projected normal vectors and \( v_i \) sum up to more than \( \pi \). If \( h_1 \) is a \( 1 \)-halfspace, then by construction of \( V_i \) we have that the angle between \( \pi(n(h_1)) \) and \( v_i \) is at most \( \frac{\pi}{2} \). Analogously, if \( h_2 \) is a \( 2 \)-halfspace, the angle between \( \pi(n(h_2)) \) and \( v_i \) is at most \( \frac{\pi}{2} \). By the choice of \( h_1 \) and \( h_2 \) we thus have \( \frac{(j_1+j_2)\pi}{k} > \pi \), which is equivalent to \( j_1 + j_2 > k \), and to \( j_1 + j_2 \geq k + 1 \), as \( j_1 \) and \( j_2 \) are integers. By definition of a \( j \)-halfspace we have

\[
\mu(h_1) + \mu(h_2) > 1 - \alpha_{k+1-j_1} + 1 - \alpha_{k+1-j_2}.
\]

Furthermore we have \( \mu(h_i) > 1 - \alpha_k \) for every \( i \in \{1, \ldots, d+1\} \), and thus

\[
\mu(h_1) + \mu(h_2) + \mu(h_3) + \ldots + \mu(h_{d+1}) > 1 - \alpha_{k+1-j_1} + 1 - \alpha_{k+1-j_2} + (d-1)(1-\alpha_k),
\]

which is equivalent to

\[
(d-1)\alpha_k + \alpha_{k+1-j_1} + \alpha_{k+1-j_2} > d + 1 - (\mu(h_1) + \ldots + \mu(h_{d+1})).
\]

As \( k+1-j_1 + k+1-j_2 = 2k+2-(j_1+j_2) \leq k+1 \), we have that \( (d-1)\alpha_k + \alpha_{k+1-j_1} + \alpha_{k+1-j_2} \leq 1 \) and thus \( \mu(h_1) + \ldots + \mu(h_{d+1}) > d \), which is a contradiction to Observation 4.

Setting \( \alpha_j = \frac{1}{kd+1} \), we get a bound for the generalized Tukey depth:

Corollary 5. Let \( \mu \) be a mass distribution in \( \mathbb{R}^d \) with \( \mu(\mathbb{R}^d) = 1 \). Then there exist \( k \) points \( p_1, \ldots, p_k \) in \( \mathbb{R}^d \) with generalized Tukey depth \( gtd_\mu(p_1, \ldots, p_k) = \frac{1}{kd+1} \).
4 Triangles

As mentioned before, the $\frac{1}{3}$-quantile and the $\frac{2}{3}$-quantile can also be interpreted as a one-dimensional simplex with the property that every halfline that contains a part of the simplex contains at least $\frac{1}{3}$ of the underlying data set and every halfline that contains the whole simplex contains at least $\frac{2}{3}$ of the underlying data set. For this interpretation, we give a generalization to two dimensions. For ease of presentation, we only give a proof for point sets instead of mass distributions and for fixed values of $\alpha$ and $\beta$.

**Theorem 6.** Let $P$ be a set of $n$ points in general position in the plane. Then there are three points $p_1$, $p_2$ and $p_3$ in $\mathbb{R}^2$ such that

1. each closed halfplane containing one of the points $p_1$, $p_2$ and $p_3$ contains at least $\frac{n}{6}$ of the points of $P$ and
2. each closed halfplane containing all of $p_1$, $p_2$ and $p_3$ contains at least $\frac{2}{3}$ points of $P$.

Note that this can also be interpreted as an instance of Theorem 1 with $\alpha_1 = \alpha_2 = \frac{1}{6}$ and $\alpha_3 = \frac{1}{2}$. However, as $\alpha_3 + \alpha_3 + \alpha_1 > 1$, the precondition of Theorem 1 does not apply.

As the proof of this result uses similar ideas as the above proofs, we only sketch the main ideas and refer the interested reader to the full version.

**Sketch of proof.** Let $C$ be the intersection of all open halfplanes containing more than $\frac{5n}{6}$ of the points of $P$. Just as in the proof of Theorem 3, condition (1) is equivalent to $p_1$, $p_2$ and $p_3$ lying in $C$. Similarly, condition (2) is equivalent to the following statement: for every halfplane $h$ containing more than $\frac{n}{2}$ of the points of $P$, $h$ contains at least one of $p_1$, $p_2$ and $p_3$. For each such $h$, let $c_h$ be the intersection of $h$ and $C$ and let $H$ be the set of all $c_h$'s that are minimal with respect to inclusion. It can be shown that among any three elements of $H$, two of them intersect. Using this property, we can then place 3 points on the boundary of $C$ such that each element of $H$ contains at least one of them: Place $p_1$ at a topmost point of the boundary of $C$. Let $h_1$ be the first element of $H$ in counterclockwise direction whose defining halfplane does not contain $p_1$. Place $p_2$ at the intersection of the defining line of $h_1$ with the boundary of $C$ that is furthest in counterclockwise direction from $p_1$. Since $h_1$ is minimal, any element of $H$ intersecting $h_1$ contains either $p_1$ or $p_2$. Further, all elements of $H$ that do not intersect $h_1$ have a common intersection, in which we place $p_3$. ◄

The general statement can be proved analogously:

**Theorem 2.** Let $\mu$ be a mass distribution in $\mathbb{R}^2$ with $\mu(\mathbb{R}^2) = 1$. Let $\alpha$ and $\beta$ be real numbers such that $0 < \alpha \leq \beta$ and $\alpha + \beta = \frac{2}{3}$. Then there is a triangle $\Delta$ in $\mathbb{R}^2$ such that

1. for each closed halfplane $h$ containing one of the vertices of $\Delta$ we have $\mu(h) \geq \alpha$ and
2. for each closed halfplane $h$ fully containing $\Delta$ we have $\mu(h) \geq \beta$.

5 Construction in the plane

In this section, we describe algorithms for constructing the points described in Theorems 3 and 6. We first observe that the convex regions defined by the intersections of the halfplanes in sets like $L$ and $R$ in the proof of Theorem 3 correspond to levels in the dual line arrangement. We use the duality $p^* = (y = kx + d) \iff p = (k, d)$ that maps a point $p$ to a line $p^*$. The $k$-level of a line arrangement is the set of points with exactly $k - 1$ lines below it and not more than $n - k$ lines above it. (It thus consists of segments of the line arrangement.) Suppose we are given $\alpha_1$ and $\alpha_2$, s.t. $0 < \alpha_1 \leq \alpha_2$ and $\alpha_1 + 2\alpha_2 = 1$. Let $U$ be the set of open halfplanes that are above their boundary lines and contain more than
Lemma 7 \cite{10, Lemma 3.2}. In an arrangement of $n$ lines, let $\gamma$ be the boundary of the convex hull of the lines on or below the $k$-level. Given the arrangement, $k$, and a point $p$, one can find the tangent to $\gamma$ passing through $p$ and touching $\gamma$ to the right of $p$ (if it exists) in time $O(n \log^2 n)$.

Lemma 8. Given an arrangement of $n$ lines and two numbers $k < l \leq n$, as well as a halfplane $h$, a line separating the $k$-level from the intersection of $h$ with the $l$-level can be found in $O(n \log^3 n)$ time, if it exists. The separating line is tangent to both level parts and, from left to right, first intersects the $k$-level and then the relevant part of the $l$-level.

Proof. Let $\gamma$ be the boundary of the convex hull of all points below the $k$-level, and let $\nu$ be the intersection of $h$ with the $l$-level. Note that $\nu$ might not be connected. Suppose we want our line to be the counterclockwise bitangent of $\gamma$ and $\nu$ (i.e., from left to right, it first intersects $\gamma$, which has no point above it, and then $\nu$). Our algorithm works by obtaining tangents to $\nu$ through points on $\gamma$. Matoušek’s $O(n \log^2 n)$ algorithm for determining the tangent to a level through a given point that is to the right of that point \cite[Lemma 3.2]{10} (our Lemma 7) also directly works for parts of a level such as $\nu$: It requires a sub-algorithm that decides in $O(n \log n)$ time whether a given line $\ell$ intersects the level (or, in our case, the partial level $\nu$). This can be done by sorting the intersection of the lines of the arrangements along $\ell$ (see also \cite[Lemma 3.1]{10}) as well as along the line bounding $h$; $\ell$ either intersects the relevant part of $\nu$, or we can compare the intersection of $h$ with $\ell$ to the intersections of $h$ with $\nu$ to determine whether there is a point of $\nu$ below $\ell$.

Suppose first we are given $\gamma$. (It requires $O(n \log^4 n)$ time though to obtain it, so we eventually get rid of this assumption.) The convex hull of a level is known to have at most $n$ vertices \cite[Lemma 2.1]{10}. For a point $p$ on $\gamma$, we can find in $O(n \log^2 n)$ time the point $q$ on $\nu$ such that the line $pq$ has no point on $\nu$ below it. We can thus find, by binary search on the $O(n)$ vertices of $\gamma$, a vertex $p$ with $q$ on $\nu$ such that $pq$ separates $\gamma$ and $\nu$. This gives an $O(n \log^4 n)$ time algorithm for obtaining the bitangent. To improve on that bound, we need to get rid of the explicit construction of $\gamma$ to find the tangents to $\nu$.

To this end, we consider Matoušek’s algorithm for constructing the convex hull boundary $\gamma$ of a level and compute only the relevant part (see \cite[Section 4]{10}). In particular, the algorithm works by finding, for a constant $c$ and two vertical lines, $(c - 1)$ further vertical lines between
Extending the Centerpoint Theorem to Multiple Points

Figure 3 A counterclockwise bitangent (brown, dash-dotted) between the $\lceil \frac{2n}{5} \rceil$-level (blue) and the $\lfloor \frac{4n}{5} \rfloor$-level (red) of an arrangement of seven lines (left). The primal point configuration is shown to the right; there, the orange region corresponds to the $\lceil \frac{n}{5} \rceil$-hull $C$, and the hatched green region corresponds to $D_U$. Observe that there can be vertices of $D_U$ outside of $C$.

the given ones such that there are at most $n^2/c$ crossings of the arrangement between two of these verticals. This can be done in $O(n)$ time (as described in [11]). The tangents on $\gamma$ at the intersection points with the vertical lines can be computed in $O(n \log^3 n)$ time [10, Lemma 3.3]. It is shown in [10] that, when choosing $c = 64$, there are at most $n/2$ lines of the arrangement relevant for the construction of $\gamma$ between two such vertical lines, and these lines can be found in $O(n)$ time. The original algorithm proceeds recursively within each interval defined by two neighboring vertical lines after removing the non-relevant lines. In our adaption, however, we find the interval that contains the point $p$ on $\gamma$ such that a tangent to $\gamma$ through the vertex $p$ with $q$ on $\nu$ such that $pq$ separates $\gamma$ and $\nu$. (We do this by considering the tangent to $\gamma$ at each of the constant number of intersection of a vertical line with $\gamma$.) When we have found this interval, we can prune $n/2$ of the lines and recurse inside this interval. Note, however, that we cannot prune the set of lines when looking for a tangent to $\nu$. Thus, in each recursive call, we need $O(n \log^2 n)$ time for computing the tangent. As the recursion depth is $O(\log n)$, this amounts to $O(n \log^3 n)$ in total. Also, for $n_i$ lines during the $i$th recursion, we need $O(n_i \log^3 n_i) \subseteq O(n \log^3 n)$ time for determining the intervals. As $n_i$ decreases geometrically, this also amounts to $O(n \log^3 n)$. This is the total running time for finding the bitangent, as claimed.

We call such a line the counterclockwise bitangent of the two subsets of the plane (i.e., the intersection with the region not above it has smaller $x$-coordinate than the intersection with the region not below it). Note that by mirroring the plane horizontally or vertically, the lemma also provides other types of bitangents. Figure 3 shows an example.

Theorem 9. Given a set $P$ of $n$ points in the plane, two points satisfying the conditions of Theorem 3 can be constructed in time $O(n \log^3 n)$.

Proof. To find a point $p_1$ in the intersection of $C$ and $D_U$, observe first that we can restrict our attention in the dual to the convex hull of the points above the $\lceil (1 - \alpha_1)n \rceil$-level of the dual line arrangement. This is because any primal line with more than $(1 - \alpha_1)n$ points above it (which corresponds to a dual point below the $\lceil \alpha_1 n \rceil$-level) also defines a halfplane in $U$. A point in the intersection of $D_U$ and $C$ thus corresponds to a line on or above the $\lceil \alpha_2 n \rceil$-level and on or below the $\lceil (1 - \alpha_1)n \rceil$-level. We find a bitangent to these two levels in $O(n \log^3 n)$ time using Lemma 8 (with $h = \mathbb{R}^2$). The primal point of this line is $p_1$; see the point indicated by the brown box in Figure 3 (right). We obtain $p_2$ analogously.
Theorem 10. Three points as described in Theorem 6 can be computed in time $O(n \log^3 n)$.

Proof. Consider the dual line arrangement of the point set. The points $p_1, p_2, p_3$ duallyize to three lines $p^*_1, p^*_2, p^*_3$ that are between the $\lceil \frac{n}{6} \rceil$-level and the $\lfloor \frac{5n}{6} \rfloor$-level of the arrangement s.t. every point on the middle level has at least one of these lines above it and one of these lines below it. (We assume for simplicity that $n$ is odd and the middle level is the $\lfloor \frac{n}{2} \rfloor$-level of the arrangement; if $n$ is even, one has to consider the points between the $\frac{n}{2}$-level and the $(\frac{n}{2} + 1)$-level.) Theorem 6 asserts that such lines exist, and its proof tells us that we can choose one of these lines to be an arbitrary tangent of one of the levels not intersecting the interior of the other one. We denote by $\gamma_b$ and $\gamma_t$ the convex hull boundaries of the points on or below the $\lceil \frac{n}{6} \rceil$-level and of the points on or above the $\lfloor \frac{5n}{6} \rfloor$-level, respectively.

We let $p^*_1$ be the clockwise bitangent of $\gamma_b$ and $\gamma_t$, which we can obtain in $O(n \log^3 n)$ time using Lemma 8. For simplicity of explanation, we also compute the counterclockwise bitangent $\ell$. (This step may be omitted in an actual implementation, but assuming it to be given facilitates the explanation and does not change the asymptotic running time.)

The line $p^*_1$ intersects the middle level of the arrangement. Let $\mu_1$ be the parts of the middle level below $p^*_1$, and $\mu_2$ be the part above it. Note that each of these parts may be disconnected. Using Lemma 8, we search for the counterclockwise bitangent between $\gamma_b$ (or, equivalently, the $\lceil \frac{n}{6} \rceil$-level) and $\mu_1$ (which is the intersection of the middle level with a halfspace defined by $p^*_1$) in $O(n \log^3 n)$ time. If it exists, and its intersection point with $\gamma_b$ is between the intersections of $\gamma_b$ with $p^*_1$ and $\ell$, we choose this line to be $p^*_2$. Otherwise, we continue our search on $\gamma_t$ in the same way (i.e., we look for the counterclockwise bitangent between $\gamma_t$ and $\mu_1$). The line $p^*_3$ can be found in an analogous manner.

Conclusion

We proposed a generalization of quantiles in higher dimensions based on a generalization of Tukey depth to multiple points. Our bounds and algorithms seem merely being a first step in this direction and we can identify several interesting open problems. Except for special cases of Theorem 1, we do not believe that our bounds are tight and particularly expect significantly better bounds in higher dimensions. Naturally, there are many other range spaces for which this problem could be considered, e.g., convex sets, like in [5].
From an algorithmic point of view, the bottleneck for the running time of our approach is Lemma 8. The current methods result in $O(n \log^3 n)$ time. While solutions to such kinds of problems can usually only be verified in $\Theta(n \log n)$ time (see, e.g., [2, 16]), a linear-time algorithm, like for centerpoints [8], is conceivable. For arbitrarily many points, it seems tedious but doable to apply similar approaches as in the proof of Theorem 9. Is there a good bound on the running time independent of the size of $|Q|$?

References


Approximate Query Processing over Static Sets and Sliding Windows

Ran Ben Basat
Harvard University, Cambridge, USA
ran@seas.harvard.edu

Seungbum Jo
University of Siegen, Germany
Seungbum.Jo@uni-siegen.de

https://orcid.org/0000-0002-8644-3691

Srinivasa Rao Satti
Seoul National University, South Korea
ssrao@cse.snu.ac.kr

https://orcid.org/0000-0003-0636-9880

Shubham Ugare
IIT Guwahati, Guwahati, India
ugare.dipak@iitg.ac.in

Abstract

Indexing of static and dynamic sets is fundamental to a large set of applications such as information retrieval and caching. Denoting the characteristic vector of the set by $B$, we consider the problem of encoding sets and multisets to support approximate versions of the operations $\text{rank}(i)$ (i.e., computing $\sum_{j \leq i} B[j]$) and $\text{select}(i)$ (i.e., finding $\min\{p : \text{rank}(p) \geq i\}$) queries. We study multiple types of approximations (allowing an error in the query or the result) and present lower bounds and succinct data structures for several variants of the problem. We also extend our model to sliding windows, in which we process a stream of elements and compute suffix sums. This is a generalization of the window summation problem that allows the user to specify the window size at query time. Here, we provide an algorithm that supports updates and queries in constant time while requiring just $(1 + o(1))$ factor more space than the fixed-window summation algorithms.

2012 ACM Subject Classification Theory of computation → Data compression

Keywords and phrases Streaming, Algorithms, Sliding window, Lower bounds

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.54

Related Version A full version of the paper is available at [1], https://arxiv.org/abs/1809.05419.

1 Introduction

Given a bit-string $B[1 \ldots n]$ of size $n$, one of the fundamental and well-known problems proposed by Jacobson [12], is to construct a space-efficient data structure which can answer $\text{rank}$ and $\text{select}$ queries on $B$ efficiently. For $b \in \{0, 1\}$, these queries are defined as follows.

1 Supported by the Zuckerman Foundation, the Technion Hiroshi Fujiwara cyber security research center, and the Israel Cyber Directorate.

2 The author of this paper is supported by the DFG research project LO748/11-1.
A bit vector supporting a subset of these operations is one of the basic building blocks in the design of various succinct data structures. Supporting these operations in constant time, with close to the optimal amount of space, both theoretically and practically, has received a wide range of attention [13, 15, 16, 17, 19]. Some of these results also explore trade-offs that allow more query time while reducing the space.

We also consider related problems in the streaming model, where a quasi-infinite sequence of integers arrives, and our algorithms need to support the operation of appending a new item to the end of the stream. For \( i \in \{1, \ldots, n\} \), let \( S_i \) be the sum of the last \( i \) integers. Here, \( n \) is the maximal suffix size we support queries for. For streaming, we consider processing a stream of elements, and answering two types of queries, *suffix sum* (ss) and *inverse suffix sum* (iss), defined as:

- ss\((i, n)\): returns \( S_i \) for any \( 1 \leq i \leq n \).
- iss\((i, n)\): returns the smallest \( j \), \( 1 \leq j \leq n \), such that \( ss(j, n) \geq i \).

In this paper, our goal is to obtain space efficient data structures for supporting a few relaxations of these queries efficiently using an amount of space below the theoretical minimum (for the unrelaxed versions), ideally. To this end, we define approximate versions of rank and select queries, and propose data structures for answering approximate rank and select queries on multisets and bit-strings. We consider the following approximate queries with an additive error \( \delta > 0 \).

- rank\(_A\((i, B, \delta)\): returns any value \( r \) which satisfies rank\(_B\((i - \delta, B) < r \leq \text{rank}_B(i, B) \). If rank\(_B\((i - \delta, B) = \text{rank}_B(i, B) \), then \text{rank\(_A\((i, B, \delta)) = \text{rank}_B(i, B) \).}
- drank\(_A\((i, B, \delta)\): returns any position \( p \) which satisfies select\(_B\((i - \delta, B) < p \leq \text{select}_B(i, B) \).
- select\(_A\((i, B, \delta)\): returns any position \( p \) which satisfies select\(_B\((i, B) - \delta < p \leq \text{select}_B(i, B) \).
- ss\(_A\((i, n, \delta)\): returns any value \( r \) which satisfies ss\((i, n) - \delta < r \leq \text{ss}(i, n) \).
- iss\(_A\((i, n, \delta)\): returns any value \( r \) which satisfies iss\((i - \delta, n) < r \leq \text{iss}(i, n) \).

We propose data structures for supporting approximate rank and select queries on bit-strings efficiently. Our data structures use less space than that is required to answer the exact queries and most of data structures use optimal space. We also propose a data structure for supporting ss\(_A\) and iss\(_A\) queries on binary streams while supporting updates efficiently. Finally, we extend some of these results to the case of larger alphabets. For all these results, we assume the standard word-RAM model [14] with word size \( \Theta(\lg n) \) if it is not explicitly mentioned.

### 1.1 Previous work

**Rank and Select over bit-strings.** Given a bit-string \( B \) of size \( n \), it is clear that at least \( n \) bits are necessary to support rank and select queries on \( B \). Jacobson [12] proposed a data structure for answering rank queries on \( B \) in constant time using \( n + o(n) \) bits. Clark and Munro [5] extended it to support both rank and select queries in constant time with \( n + o(n) \) bits. For the case when there are \( m \) 1’s in \( B \), at least \( B(n, m) \) bits\(^3\) are necessary.

\(^3\) \( B(n, m) = \lfloor \frac{n}{m} \rfloor \) bits is the information-theoretic lower bound on space for storing a subset of size \( m \leq n \) from the universe \( \{1, 2, \ldots, n\} \).
to support rank and select on $B$. Raman et al. [19] proposed a data structure that supports both operations in constant time while using $\mathcal{B}(n, m) + o(n) + O(\lg\lg m)$ bits. Golynski et al. [10] gave an asymptotically optimal time-space trade-off for supporting rank and select queries on $B$. A slightly related problem of approximate color counting has been considered in El-Zein et al. [7].

**Algorithms that Sum over Sliding Windows.** Our ss queries for streaming are a generalization of the problem of summing over sliding windows. That is, window summation is a special case of the suffix sum problem where the algorithm is always asked for the sum of the last $i \leq n$ elements. Approximating the sum of the last $n$ elements over a stream of integers in $\{0, 1, \ldots, \ell\}$, was first introduced by Datar et al. [6]. They proposed a $(1 + \varepsilon)$ multiplicative approximation algorithm that uses $O\left(\varepsilon^{-1} \left(\frac{\ell^2}{n} + \ell \cdot (\log n + \log \log \ell)\right)\right)$ bits and operates in amortized time $O(\log \ell / \log n)$ or $O(\log (\ell \cdot n))$ worst case. In [8], Gibbons and Tirthapura presented a $(1 + \varepsilon)$ multiplicative approximation algorithm that operates in constant worst case time while using similar space for $\ell = n^{O(1)}$. [3] studied the potential memory savings one can get by replacing the $(1 + \varepsilon)$ multiplicative guarantee with a $\delta$ additive approximation. They showed that $\Theta\left((\ell \cdot n / \delta + \log n)\right)$ bits are required and sufficient. Recently, [2] showed the potential memory saving of a bi-criteria approximation, which allows error in both the sum and the time axis, for sliding window summation. [4] looks at a generalization of the ssA queries to general alphabet, where at query time we also receive an element $x$ and return an estimate for the frequency of $x$ in the last $i$ elements.

It is worth mentioning that these data structures do allow computing the sum of a window whose size is given at the query time. Alas, the query time will be slower as they do not keep aggregates that allow quick computation. Specifically, we can compute a $(1 + \varepsilon)$ multiplicative approximation in $O(\varepsilon^{-1} \log (\ell \varepsilon))$ time using the data structures of [6] and [8]. We can also use the data structure of [3] for an additive approximation of $\delta$ in $O(n \ell / \delta)$ time.

### 1.2 Our results

In this paper, we obtain the following results for the approximate rank, select, ss and iss queries with additive error. Let $B$ be a bit-string of size $n$.

1. **rank and select queries with additive error $\delta$.** In this case, we first show that $\lceil n / \delta \rceil$ bits are necessary for answering $\text{drank}_{A_1}$ and $\text{select}_{A_1}$ queries on $B$ and propose a $(\lceil n / \delta \rceil + o(n / \delta))$-bit data structure that supports $\text{drank}_{A_1}$ and $\text{select}_{A_1}$ queries on $B$ in constant time. For the

### Table 1

<table>
<thead>
<tr>
<th>Query</th>
<th>Space (in bits)</th>
<th>Query time</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lower bounds</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{drank}<em>{A_1}, \text{select}</em>{A_1}$</td>
<td>$\left\lceil n / \delta \right\rceil$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{drank}<em>{A_1}, \text{select}</em>{A_1}$</td>
<td>$\mathcal{B}(\lceil n / \delta \rceil, \lceil m / \delta \rceil)$</td>
<td>$\delta$, additive</td>
<td></td>
</tr>
<tr>
<td>$\text{rank}<em>{A_1}, \text{select}</em>{A_1}$</td>
<td>$\lceil n / \delta \rceil \log \delta$</td>
<td>$O((\log(n / \delta)) \log^{O(1)}(\delta))$</td>
<td>$\Omega(\log \log n)$</td>
</tr>
<tr>
<td>Upper bounds</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{drank}<em>{A_1}, \text{select}</em>{A_1}$</td>
<td>$n / \delta + o(n / \delta)$</td>
<td>$O(1)$</td>
<td>$\delta$, additive</td>
</tr>
<tr>
<td>$\text{rank}_{A_1}$</td>
<td>$n / \delta \log \delta + o(n / \delta) \log \delta$</td>
<td>$O(n / \delta, n)$</td>
<td></td>
</tr>
<tr>
<td>$\text{select}_{A_1}$</td>
<td>$(n / \delta) \log \delta + o((n / \delta) \log \delta)$</td>
<td>$t(n / \delta, n)$</td>
<td></td>
</tr>
</tbody>
</table>
Table 2 Comparison of data structures for \(ss\) queries over stream of integers in \(\{0, \ldots, \ell\}\). All works can answer fixed-size window queries (where \(i \equiv n\) in \(O(1)\) time. Worst case times are specified.

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Guarantee</th>
<th>Space (in bits)</th>
<th>Update Time</th>
<th>Query time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGIM02 ([6])</td>
<td>((1 + \epsilon))-multiplicative</td>
<td>(O(\epsilon^{-1} \lg n \lg (\lg \ell)))</td>
<td>(O(\lg (\ell n)))</td>
<td>(O(\epsilon^{-1} \lg (\ell n)))</td>
</tr>
<tr>
<td>GT02 ([8])</td>
<td>((1 + \epsilon))-multiplicative</td>
<td>(O(\epsilon^{-1} \lg^* (\ell n)))</td>
<td>(O(1))</td>
<td>(O(\epsilon^{-1} \lg (\ell n)))</td>
</tr>
<tr>
<td>BEFK16 ([3])</td>
<td>(\delta)-additive, for (\delta = \Omega(\ell))</td>
<td>(\Theta(\ell \cdot n/\delta + \lg n))</td>
<td>(O(1))</td>
<td>(O(\ell \cdot n/\delta))</td>
</tr>
<tr>
<td>BEFK16 ([3])</td>
<td>(\delta)-additive, for (\delta = o(\ell))</td>
<td>(\Theta(n \lg (\ell/\delta)))</td>
<td>(O(1))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>This paper</td>
<td>(\delta)-additive</td>
<td>Same as in ([3])</td>
<td>(O(1))</td>
<td>(O(1))</td>
</tr>
</tbody>
</table>

case when there are \(m\) 1’s in \(B\), we show that \(B([n/\delta], [m/\delta])\) bits are necessary for answering \(\text{rankA}_1\) and \(\text{selectA}_1\) queries on \(B\), and obtain \(B([n/\delta], [m/\delta]) + o(n/\delta)\)-bit data structure that supports \(\text{rankA}_1\) and \(\text{selectA}_1\) queries on \(B\) in constant time. For \(\text{rankA}_1\) and \(\text{selectA}_1\) queries on \(B\), we show that \(\lfloor n/2\delta \rfloor \lg \delta\) bits are necessary for answering both queries, and obtain an \((n/\delta) \lg \delta + o((n/\delta) \lg \delta)\)-bit data structure that supports \(\text{rankA}_1\) queries in \(O(1)\) time, and \(\text{selectA}_1\) queries in \(O(\min \{\lg \lg (n/\delta) \lg \lg n / \lg \lg n, \sqrt{\lg (n/\delta)} / \lg (n/\delta)\})\) time. Furthermore, we show that there exists an additive error \(\delta\) such that any \(O((n/\delta) \lg (O(1))\)-bit data structure requires at least \(\Omega(\lg \lg n)\) time to answer \(\text{selectA}_1\) queries on \(B\).

Using the above data structures, we also obtain data structures for answering approximate \(\text{rank}\) and \(\text{select}\) queries on a given multiset \(S\) from the universe \(U = \{1, 2, \ldots, n\}\) with additive error \(\delta\), where \(\text{rank}(i, S)\) query returns the value \(|\{j \in S | j \leq i\}|\), and \(\text{select}(i, S)\) query returns the \(i\)-th smallest element in \(S\). We consider two different cases: (i) \(\text{rankA}, \text{drankA}\) \(\text{selectA}\), and \(\text{dselectA}\) queries when \(|S| = m\), and (ii) \(\text{drankA}\) and \(\text{selectA}\) queries when the frequency each elements in \(S\) is at most \(\ell\). Furthermore for case (ii), we first show that at least \(\lfloor n/\lfloor \ell/\delta \rfloor \rfloor \lg (\max (\lfloor \ell/\delta \rfloor, 1) + 1)\) bits are necessary for answering \(\text{drankA}\) queries, and obtain an optimal space structure that supports \(\text{drankA}\) queries in constant time, and an asymptotically optimal space structure that supports both \(\text{drankA}\) and \(\text{selectA}\) queries in constant time when \(\ell = O(\delta)\).

We also consider the \(\text{drankA}\) and \(\text{selectA}\) queries on strings over large alphabets. Given a string \(A\) of length \(n\) over the alphabet \(\Sigma = \{1, 2, \ldots, \sigma\}\) of size \(\sigma\), we obtain an \((2n/\delta \lg (\sigma + 1) + o((n/\delta) \lg (\sigma + 1)\))-bit data structure that supports \(\text{drankA}\) and \(\text{selectA}\) on \(A\) in \(O(\lg \lg \sigma)\) time. We summarize our results for bit-strings in Table 1.

2. \(ss\) and \(iss\) queries with additive error \(\delta\). We first consider a data structure for answering \(ss\) and \(iss\) queries on binary stream, i.e., all integers in the stream are 0 or 1. For exact \(ss\) and \(iss\) queries on the stream, we propose an \(n + o(n)\)-bit data structure for answering those queries in constant time while supporting constant time updates whenever a new element arrives from the stream. This data structure is obtained by modifying the data structure of Clark and Munro \([5]\) for answering \(\text{rank}\) and \(\text{select}\) queries on bit-strings. Using the above structure, we obtain an \((n/\delta + o(n/\delta) + O(\lg n))\)-bit structure that supports \(ssA\) and \(issA\) queries on the stream in constant time while supporting constant time updates. Since at least \(n/\delta\) bits are necessary for answering \(\text{drankA}_1\) (or \(\text{selectA}_1\)) queries on bit-strings, and \(\lg n\) bits are necessary for answering \(ss(n, n)\) queries \([3]\), the space usage of our data structure is succinct (i.e., optimal up to lower-order terms) when \(n/\delta = \omega(\lg n)\), and asymptotically optimal otherwise.

We then consider the generalization that allows integers in the range \(\{0, 1, \ldots, \ell\}\), for some \(\ell \in \mathbb{N}\). First, we present an algorithm that uses the optimal \(n \lg (\ell + 1)(1 + o(1))\) bits for exact suffix sums. Then, we provide a second algorithm that uses \(|n/\lfloor \delta/\ell \rfloor\| \lg (\max (\lfloor \ell/\delta \rfloor, 1) + 1) + o(1)) + O(\lg n)\) bits for solving \(ssA\). Specifically, our data structure is succinct when \(n/\delta = \omega(\lg n/\ell)\), and is asymptotically optimal otherwise, and improves the query time of \([3]\) while using the same space. Table 2 presents this comparison.
2 Queries on bit-strings

In this section, we first consider the data structures for answering approximate rank and select queries on bit-strings and multisets. We also show how to extend our data structures on static bit-strings to the sliding windows on binary streams, for answering approximate ss and iss queries.

2.1 Approximate rank and select queries on bit-strings

We now consider the approximate rank and select queries on bit-strings with additive error \( \delta \). We only show how to support \( \text{rank}_A, \text{drank}_A, \text{dselect}_A \), and \( \text{select}_A \) queries. To support \( \text{rank}_A, \text{drank}_A, \text{dselect}_A \), and \( \text{select}_A \) queries, one can construct the same data structures on the bit-wise complement of the original bit-string. We first introduce a few previous results which will be used in our structures. The following lemmas describe the optimal structures for supporting rank and select queries on bit-strings.

- **Lemma 1 ([5]).** For a bit-string \( B \) of length \( n \), there is a data structure of size \( n + o(n) \) bits that supports \( \text{rank}_0, \text{rank}_1, \text{select}_0, \) and \( \text{select}_1 \) queries in \( O(1) \) time.

- **Lemma 2 ([19]).** For bit-string \( B \) of length \( n \) with \( m \) 1’s, there is a data structure of size

  (a) \( B(n,m) + o(m) \) bits that supports \( \text{select}_1 \) query in \( O(1) \) time, and

  (b) \( B(n,m) + o(n) \) bits that supports \( \text{rank}_0, \text{rank}_1, \text{select}_0, \) and \( \text{select}_1 \) queries in \( O(1) \) time.

We use results from [11] and [18], which describe efficient data structures for supporting the following queries on integer arrays. For a standard word-RAM model with word size \( O(\log U) \) bits, let \( A \) be an array of \( n \) non-negative integers. For \( 1 \leq i \leq n \) and any non-negative integer \( x \), (i) \( \text{sum}(i) \) returns the value \( \sum_{j=1}^{i} A[j] \), and (ii) \( \text{search}(x) \) returns the smallest \( i \) such that \( \text{sum}(i) > x \). We use the following function to state the running time of some of the (Searchable Partial Sum) queries in the lemma below, and in the rest of the paper.

\[
S(n, U) = \begin{cases} 
O(1) & \text{if } n = \text{polylog}(U) \\
O(\min \{ \log \log n \log U / \log \log U, \sqrt{\log n / \log \log n} \}) & \text{otherwise}
\end{cases}
\]

- **Lemma 3 ([11], [18]).** An array of \( n \) non-negative integers, each of length at most \( \alpha \) bits, can be stored using \( \alpha(n) \) bits, to support \( \text{sum} \) queries on \( A \) in constant time, and \( \text{search} \) queries on \( A \) in \( S(n, n2^{\alpha}) \) time. Moreover, when \( \alpha = O(\log \log n) \), we can answer both queries in \( O(1) \) time.

**Supporting drankA and selectA queries.** We first consider the problem of supporting \( \text{drank}_A \) or \( \text{select}_A \) queries with additive error \( \delta \) on a bit-string \( B \) of length \( n \). We first prove a lower bound on space used by any data structure that supports either of these two queries.

- **Theorem 4.** Any data structure that supports \( \text{drank}_A \) or \( \text{select}_A \) queries with additive error \( \delta \) on a bit-string of length \( n \) requires at least \( \lceil n/\delta \rceil \) bits. Also if the bit-string has \( m \) 1’s in it, then at least \( B(\lceil n/\delta \rceil, \lfloor m/\delta \rfloor) \) bits are necessary for answering the above queries.

**Proof.** Consider a bit-string \( B \) of length \( n \) divided into \( \lceil n/\delta \rceil \) blocks \( B_1, B_2, \ldots, B_{\lceil n/\delta \rceil} \) such that for \( 1 \leq i < \lceil n/\delta \rceil \), \( B_i = B[\delta(i-1) + 1 \ldots \delta i] \) and \( B_{\lceil n/\delta \rceil} = B[\delta \lfloor (n/\delta) - 1 \rfloor + 1 \ldots n] \) (the last block may contain more than \( \delta \) bits). Let \( S \) be the set of all possible bit-strings satisfying the condition that all the bits within a block are the same (i.e., either all zeros or all ones). Then it is easy to see that \( |S| = 2^{\lfloor n/\delta \rfloor} \). We now show that any two distinct bit-strings in \( S \) will have different answers for some \( \text{drank}_A \) query (and also some \( \text{select}_A \) query). Consider
two distinct bit-strings $B$ and $B'$ in $S$, and let $i$ be the index of the leftmost block such that $B_i \neq B'_i$. Then it is easy to show that there is no value which is the answer of both $\text{drank}_{A_1}(i\delta, B, \delta)$ and $\text{drank}_{A_1}(i\delta, B', \delta)$ queries and also there is no position of $B$ which is the answer of both $\text{select}_{A_1}(j, B, \delta)$ and $\text{select}_{A_1}(j, B', \delta)$ queries, where $j$ is the number of 1’s in $B[1 \ldots i\delta]$. Thus any structure that supports either of these queries must distinguish between every element in $S$, and hence $\lceil n/\delta \rceil$ bits are necessary to answer $\text{drank}_{A_1}$ or $\text{select}_{A_1}$ queries.

For the case when the number of 1’s in the bit-string is fixed to be $m$, we choose $\lceil m/\delta \rceil$ blocks from each bit-string and make all bits in the chosen blocks to be 1’s (and the rest of the bits as 0’s). Since there are $\binom{\lceil n/\delta \rceil}{\lceil m/\delta \rceil}$ ways for select such $\lceil m/\delta \rceil$ blocks in a bit-string of length $n$, it implies that $B(\lceil n/\delta \rceil, \lceil m/\delta \rceil)$ bits are necessary to answer $\text{drank}_{A_1}$ and $\text{select}_{A_1}$ queries in this case. \hfill ▷

Now we describe a data structure for supporting $\text{drank}_{A_1}$ and $\text{select}_{A_1}$ queries in constant time, using optimal space.

**Theorem 5.** For a bit-string $B$ of length $n$, there is a data structure that uses $n/\delta + o(n/\delta)$ bits and supports $\text{drank}_{A_1}$ and $\text{select}_{A_1}$ queries with additive error $\delta$, in constant time. If there are $m$ 1’s in $B$, the data structure uses $B(n/\delta, m/\delta) + o(n/\delta)$ bits and supports the queries in $O(1)$ time.

**Proof.** We divide the $B$ into $\lceil n/\delta \rceil$ blocks $B_1, B_2, \ldots B_{\lceil n/\delta \rceil}$ such that for $1 \leq i < \lceil n/\delta \rceil$, $B_i = B[i\delta(i-1) + 1 \ldots i\delta]$ and $B_{\lceil n/\delta \rceil} = B[i\delta(\lceil n/\delta \rceil - 1) + 1 \ldots n]$. Now we define a new bit-string $B'$ of length $\lceil n/\delta \rceil$ such that for $1 \leq i \leq \lceil n/\delta \rceil$, $B'[i] = 1$ if $B_i$ contains $j\delta$-th 1 in $B$ for any $j \leq i$, and otherwise $B'[i] = 0$ (note that for any $1 \leq j \leq \lceil n/\delta \rceil$, any block of $B$ has at most one position of $j\delta$-th 1 in $B$). By Lemma 1, we can support $\text{rank}_1$ and $\text{select}_1$ queries on $B'$ in constant time, using $n/\delta + o(n/\delta)$ bits. Now we claim that $C = \delta \cdot \text{rank}_1([i/\delta]) + (i \mod \delta)B'[i/\delta]$ gives an answer of the $\text{drank}_{A_1}(i, B, \delta)$ query. Let $D = \delta \cdot \text{rank}_1([i/\delta])$, and let $d$ be the position of $D$-th 1 in $B$. From the definition of $B'$, we can easily show that if $B'[i/\delta] = 0$ or $(i \mod \delta) \neq 0$. Then there are at most $(\delta + (i \mod \delta) - 1)$ 1’s in $B[d \ldots i]$ when $(\delta + i/\delta) + 1)$ is the position of the $(D + \delta)$-th 1 in $B$, and all the values in $B[\delta[i/\delta] + 2 \ldots i]$ are 1. Also there are at least $\delta - (\delta - (i \mod \delta)) = (i \mod \delta)$ 1’s in $B[d \ldots i]$ when $(\delta + i/\delta)$ is the position of the $(D + \delta)$-th 1 in $B$ and all the values in $B[\delta[i/\delta] + (i \mod \delta) + 1 \ldots \delta](i/\delta)$ are 1. By the similar argument, we can show that one can answer the $\text{select}_{A_1}(i, B, \delta)$ query in $O(1)$ time by returning $\delta(\text{select}_1([i/\delta], B') - 1) + (i \mod \delta)$.

Finally, in the case when there are $m$ 1’s in $B$, there are at most $\lceil m/\delta \rceil$ 1’s in $B'$. Therefore by Lemma 2(b), we can support $\text{drank}_{A_1}$ and $\text{select}_{A_1}$ queries (as before) in $O(1)$ time, using $B(n/\delta, m/\delta) + o(n/\delta)$ bits. \hfill ▷

Note that in the above proof, we can answer $\text{drank}_{A_1}$ (or $\text{select}_{A_1}$) queries on $B$ using any data structure that supports $\text{rank}_1$ (or $\text{select}_1$) queries on $B'$. Thus, if $B$ is very sparse, i.e., when $B(n/\delta, m/\delta) \ll o(n/\delta)$ (in this case, the space usage of the structure of Theorem 5 is sub-optimal), one can use the structure of [17] that uses $(m/\delta)(\log(n/m)) + O(m/\delta)$ bits (asymptotically optimal space), to support $\text{drank}_{A_1}$ queries in $O(\min\{\log m, \log(n/m)\})$ time, and $\text{select}_{A_1}$ queries in constant time.

**Supporting $\text{rank}_A$ and $\text{select}_A$ queries.** Now we consider the problem of supporting $\text{rank}_{A_1}$ and $\text{select}_{A_1}$ queries with additive error $\delta$ on bit-strings of length $n$. The following theorem describes a lower bound on space.
Theorem 6 (⋆). Any data structures that supports rank$_A$ or dselect$_A$ queries with additive error $\delta$ on a bit-string of length $n$ requires at least $\lceil n/2\delta \rceil \lg \delta$ bits.

We now show that for some values of $\delta$, any data structure that uses up to a $\lg^\Theta(1)\delta$ factor more than the optimal space cannot support dselect$_A$ queries in constant time.

Theorem 7 (⋆). Any $((n/\delta)\lg^\Theta(1)\delta)$-bit data structure that supports dselect$_A$ queries with an additive error $\delta = O(n^c)$, for some constant $0 < c \leq 1$ on a bit-string of length $n$ requires $\Omega(\lg \lg n)$ query time.

The following theorem describes a simple data structure for supporting dselect$_A$ queries.

Theorem 8 (⋆). For a bit-string $B$ of length $n$, there is a data structure of size $(n/\delta)\lg \delta + o((n/\delta)\lg \delta)$ bits, which supports rank$_A$ queries on $B$ using $O(1)$ time and dselect$_A$ queries on $B$ using $\text{SPS}(n/\delta, n)$ time.

2.2 Approximate rank and select queries on multisets

In this section, we describe data structures for answering approximate rank and select queries on a multiset with additive error $\delta$. Given a multiset $S$ where each element is from the universe $U = \{1, 2, \ldots, n\}$, the rank and select queries on $S$ are defined as follows.

- rank$(i, S)$: returns the value $|\{j \in S | j \leq i\}|$.
- select$_0(i, S)$: returns the $i$-th smallest element in $S$.

We now describe efficient structures for the following two cases.

1) rank$_A$, drank$_A$, select$_A$, and dselect$_A$ queries when $|S| = m$ is fixed. We construct a new string $B'_S$ of length $[m/\delta] + n$ such that $B'_S$ only keeps every $i$-th 1 from $B_S$, for $1 \leq i \leq [n/\delta]$ (and removes all other 1’s). To answer the query drank$_A(i, S, \delta)$, we first compute select$_0(i, B'_S) - i = \text{rank}(i, S)/\delta$, and return $\text{dselect}(i, B'_S, -i)$ as the answer. It is easy to see that $\delta \cdot \text{rank}(i, S)/\delta$ is an answer to the drank$_A(i, S, \delta)$ query. Similarly, we can answer the select$_A(i, S, \delta)$ query by returning rank$_0(\text{select}_A([i/\delta], B'_S), B'_S) + 1$. We represent $B'_S$ using the structure of Lemma 2(b), which uses $B(n + [m/\delta], [m/\delta]) + o(n + [m/\delta])$ bits and supports rank$_0$, drank$_0$, select$_0$ and select$_1$ queries on $B'_S$ in constant time. Thus, both drank$_A$ and select$_A$ queries on $S$ can be supported in constant time.

For answering rank$_A$ and dselect$_A$ queries on $S$, we first construct the data structure of Theorem 8 to support dselect$_A$ queries on $B_S$. In addition, we maintain the data structure of Lemma 3 to support sum and search queries on arrays $D[1 \ldots [n + m]/\delta]$ and $E[1 \ldots [n + m]/\delta]$ which are defined as follows. For $1 \leq i \leq [n + m]/\delta$, $D[i]$ and $E[i]$ stores the number of 0’s and 1’s in the block $B_S$, respectively (as defined in the proof of Theorem 8). By Lemma 3 and Theorem 8, the total space for this data structure is $O((n'/\delta)\lg \delta)$ bits. To answer rank$_A(i, S, \delta)$, we first find the block $B_S$ of $B_S$ which contains i-th 0 by answering search$(i)$ query on $D$, and then return sum$(j - 1)$ query on $E$. To
answer \( \text{dselectA}(i,S,\delta) \), we first find the block \( B_j \) of \( B_S \) which contains an answer of the \( \text{dselectA}_1(i,B_j,\delta) \) query, and then return \( \text{sum}(j-1) \) as the answer for \( \text{dselectA}(i,S,\delta) \). Note that if \( j = 1 \), we return 0 for both queries. The total running time is \( \text{SPS}(n'/\delta,n') \) for both \( \text{rankA} \) and \( \text{dselectA} \) queries on \( S \), by Lemma 3 and Theorem 8. For special case when \( \min\{(n+m)/\delta,\delta\} = \text{polylog}(n+m) \), we can answer \( \text{rankA} \) and \( \text{dselectA} \) queries on \( S \) in constant time.

**Theorem 9**. Given a multiset \( S \) where each element is from the universe \( U = \{1,2,\ldots,n\} \) of size \( n \), any data structure that supports \( \text{drankA} \) queries on \( S \) requires at least \( [n/\lceil \delta/\ell \rceil] \log (\max ([\ell/\delta], 1) + 1) \) bits, where \( \ell \) is a bound on the maximum frequency of each element in \( S \).

We describe a data structure which answers \( \text{drankA} \) and \( \text{selectA} \) queries on \( S \) in \( O(1) \) time. For \( \text{drankA} \) queries, it uses the optimal space. The details are omitted due to space limitation.

### 2.3 Approximate ss and iss queries on binary streams

In this section, we consider a data structure for answering \( \text{ssA} \) and \( \text{issA} \) queries on a binary stream. We first show how to modify the data structure of the Lemma 1, for answering \( \text{ss}(i,n) \) and \( \text{iss}(i,n) \) queries in constant time using \( n + o(n) \) bits, while supporting updates in constant time. We break the stream into *frames*, which is \( n \)-bit consecutive elements in the stream. Since one can construct a data structure of Lemma 1 in online [5], it is easy to show that we can answer \( \text{ss} \) and \( \text{iss} \) queries in constant time using \( 2n + o(n) \) bits while supporting constant-time updates by maintaining two data structure of Lemma 1 such as one for the current frame and other for the previous frame of the stream. To make this data structure using \( n + o(n) \) bits, we construct a data structure of Lemma 1 on the new frame while replacing the oldest part of the data structure constructed on the previous frame. The details of the succinct data structure are omitted due to space limitation.

Next, we consider a data structure for answering \( \text{ssA}(i,n,\delta) \) and \( \text{issA}(i,n,\delta) \) queries on the binary stream in constant time using \( [n/\delta] + O(\log n) + o(n/\delta) \) bits. We first split each frame \( f = f_1 \ldots f_n \) into \( [n/\delta] \) chunks \( g_1 \ldots g_{[n/\delta]} \) such that for \( 1 \leq i \leq [n/\delta] \), \( g_i = 1 \) if and only if \( f_{(i-1)\delta+1} \ldots f_{\text{min}(n,i\delta)} \) contains \( j \delta \)-th 1 in \( f \) for any integer \( j \leq i \). Now consider a (virtual) binary stream of \( g_i \)'s. Then we can construct an \( [n/\delta] + o(n/\delta) \)-bit data structure for answering \( \text{ss}(i,n) \), \( \text{iss}(i,n) \) queries in constant time while supporting constant-time updates on the such stream (In the rest of this section, all of \( \text{ss} \) and \( \text{iss} \) queries are answered on the virtual stream). We also maintain \( c \) and \( t_c \), which stores the number of 1’s in the current frame and chunk of the stream respectively. Finally, we maintain an value \( t \) which is an index of the last-arrived element in the current frame. All these additional values can be stored using \( O(\log n) \) bits.

When \( f_i \) is arrived, We first increase \( c \) and \( t_c \) by 1 if \( f_i = 1 \) if \( f \mod \delta = 0 \) or \( t = n \), we send 1 to the virtual stream if there is an integer \( j \leq t \) such that \( c - t_c \leq j \delta \leq c \), and send 0 to the virtual stream otherwise. After that, we update the data structure which supports \( \text{ss} \) and \( \text{iss} \) queries on the virtual stream, and reset \( t_c \) to zero (if \( t = n \), we also reset \( c \) to zero). Since we can update the data structure on the virtual stream in constant time, the above procedure can be done in constant time. Now we describe how to answer \( \text{ssA} \) and \( \text{issA} \) queries.
ssA queries: To answer the ssA(i, S, δ) query, we return 0 if i ≤ δ. If not, let f′ i be the \((i - (t \mod \delta))/\delta\)-th last element in the virtual stream. Then we return 
\[ tc + \delta sA((i - (t \mod \delta))/\delta, \lceil n/\delta \rceil) + (i - (t \mod \delta) \mod \delta) f′ i, \]
which gives an answer of the ssA(i, n, δ) query by the same argument as the proof of Theorem 5.

issA queries: To answer the issA(i, n, δ) query, we return \[ n - (t - (t \mod \delta)) \] if i ≤ tc. Otherwise, we return \[ n - ((i - tc)/\delta + ((i - tc) \mod \delta)) \] by the same argument as the proof of Theorem 5.

Since ss and iss queries on the virtual stream take O(1) time, we can answer both ssA and issA queries on the stream in O(1) time. Thus we obtain the following theorem.

**Theorem 10.** For a binary stream, there exists a data structure that uses \(\lceil n/\delta \rceil + O(\lg n) + o(\lceil n/\delta \rceil)\) bits and supports ssA and issA queries on the stream with additive error δ, in constant time. Also, the structure supports updates in constant time.

Comparing to the lower bound of Theorem 4 for answering drankA and selectA queries on bit-strings (this also gives a lower bound for answering ssA and issA queries), the above data structure takes \(\Omega(n/\delta)\) bits when \(n/\delta = o(\lg n)\). However in the sliding window of size \(n\), at least \(\lceil \lg n \rceil\) bits are necessary [3] for answering ssA queries even the case when \(i\) is fixed to \(n\). Therefore the data structure of Theorem 10 supports ssA and issA queries with optimal space when \(n/\delta = \omega(\lg n)\), and asymptotically optimal otherwise.

### 3 Queries on strings over large alphabet

In this section, we consider non-binary inputs. First, we look at general alphabet and derive results for approximate rank and select. Then we consider suffix sums over integer streams.

#### 3.1 drankA and selectA queries on strings over general alphabet

Let \(A\) be a string of length \(n\) over the alphabet \(\Sigma = \{1, 2, \ldots, \sigma\}\) of size \(\sigma\). Then, for \(1 \leq j \leq \sigma\), the query \(\text{rank}_j(i, A)\) returns the number of \(j\)'s in \(A[1 \ldots i]\), and the query \(\text{select}_j(i, A)\) returns the position of the \(i\)-th \(j\) in \(A\) (if it exists). Similarly, the queries drank\(A\)_\(j\)(i, A, δ) and select\(A\)_\(j\)(i, A, δ) are defined analogous to the queries drankA and selectA on bit-strings. One can easily show that at least \(\lceil n/\delta \rceil \lg \sigma\) bits are necessary to support drankA and selectA queries on \(A\), by extending the proof of Theorem 4 to strings over larger alphabets. In this section, we describe a data structure that supports drankA and selectA queries on \(A\) in \(O(\lg \lg \sigma)\) time, using twice the optimal space. We make use of the following result from [9] for supporting rank and select queries on strings over large alphabets. We now use the following lemma to prove our main result for the section.

**Lemma 11 ([9]).** Given a string of length \(n\) over the alphabet \(\Sigma = \{1, 2, \ldots, \sigma\}\), one can support \(\text{rank}_j\) queries in \(O(\lg \lg \sigma)\) time and \(\text{select}_j\) queries in \(O(1)\) time, using \(n \lg \sigma + o(n \lg \sigma)\) bits, for any \(1 \leq j \leq \sigma\).

The following theorem shows we can construct a simple data structure for supporting drank\(A\)_\(j\) and select\(A\)_\(j\) queries on \(A\) using the above lemma.

**Theorem 12 (⋆).** Let \(A\) be a string of length \(n\) over the alphabet \(\Sigma = \{1, 2, \ldots, \sigma\}\). Then for any \(1 \leq j \leq \sigma\), one can support drank\(A\)_\(j\) and select\(A\)_\(j\) queries in \(O(\lg \lg \sigma)\) time using \(2n/\delta \lg (\sigma + 1) + o((n/\delta) \lg (\sigma + 1))\) bits.
3.2 Supporting ssA queries over non-binary streams

In this section, we consider the problem of computing suffix sums over a stream of integers in \(\{1, 2, \ldots, \ell\}\). This generalizes the result of the Theorem 10 for ssA. For such streams, one can use ssA binary search to solve ssA, while a constant time ssA queries are left as future work. Specifically, we show a data structure that requires \(|n/\lfloor \delta/\ell \rfloor| \lg (\max (\lfloor \ell/\delta \rfloor, 1) + 1) (1 + o(1)) + O(\lg n)\); i.e., it requires \(1 + o(1)\) times as many bits as the static-case lower bound of Theorem 9 when \(\delta = o(\ell \cdot n/\lg n)\).

We note that this model was studied in \([3, 6, 8]\) for window-sum queries. That is, our work generalizes this model to allow the user to specify the window size \(\delta\) whereas previous works only considered the sum of the last \(\ell\) elements. In fact, all previous data structure implicitly supports ssA queries but with slower run time. \([8, 6]\) requires \(O(e^{-1} \lg (\ell \nu))\) time to compute a \((1 + \epsilon)\) approximation for the sum of the last \(n\) elements while \([3]\) needs \(O(\ell \cdot n/\delta)\) for a \(\delta\)-additive one. Here, we show how to compute a \(\delta\)-additive error for the sum of the last \(i \leq n\) elements in constant time for both updates and queries.

**Exact ss queries.** En route to ssA, we first discuss how to compute an exact answer for suffix sums queries. It is known, even for fixed window sizes, that one must use \(n \lg (\ell + 1)\) bits for tracking the sum of a sliding window \([3]\). Here, we show how to compute exact ssA using succinct space of \(n \lg (\ell + 1) (1 + o(1))\) bits.

We start by discussing why the current approaches cannot work for a large \(\ell\). If we use sub-blocks of size \(\Theta(\lg n)\) as in \([5, 12]\), then the lookup table will require \((\ell + 1)^{\Theta(\lg n)} = n^{\Theta(\lg (\ell + 1))}\) bits, which is not even asymptotically optimal for non-constant \(\ell\) values. While one may think that this is solvable by further breaking the sub-blocks into sub-sub-blocks, sub-sub-sub-blocks, etc., it is not the case. To see this, consider a lookup table for sequences of length 2. Then its space requirement will be \((\ell + 1)^2\) bits. If \(\ell\) is large (say, \(\ell \geq n\)) then this becomes \(\Omega(n(\ell + 1)^2)\), which is not even asymptotically optimal.

\[\textbf{Theorem 13 (\star). There exists a data structure that requires } n \lg (\ell + 1) (1 + o(1)) \text{ bits and support constant time (exact) suffix sums queries and updates.}\]

**General ssA queries.** Here, we consider the general problem of computing ssA (i.e., up to an additive error of \(\delta\)). Intuitively, we apply the exact solution from the previous section on a compressed stream that we construct on the fly. A simple approach would be to divide the streams into consecutive chunks of size \(\max ([\mu], 1) = \max ([\delta/\ell], 1)\) and represent each chunk’s sum as an input to an exact suffix sum algorithm. However, this fails to achieve succinct space. For example, summing \([\delta/\ell]\) integers requires \(O([\delta/\ell] \lg (\ell + 1)) = \Omega(\lg \ell)\) bits. However, \(\lg \ell\) bits may be asymptotically larger than the \([n/\lfloor \mu \rfloor] \lg (\max ([1/\mu], 1) + 1)\) bits lower bound of Theorem 9.

We alleviate this problem by **rounding** the arriving elements. Namely, when adding an input \(x \in \{0, 1, \ldots, \ell\}\), we first round its value to \(\text{Round}_b(x) \triangleq 2^{-b} \beta \cdot \left\lfloor \frac{2^b}{\ell} \right\rfloor\) so it will require \(b \triangleq \lfloor \lg (n/\mu) + \lg \lg n \rfloor\) bits. The rounding allows us to sum elements in a chunk (using a variable denoted by \(r\)), but introduces a rounding error. To compensate for the error, we both consider a smaller chunks; namely, we use chunks of size \(\nu \triangleq \max \{[\mu \cdot (1 - 1/\lg n)], 1\}\). We also consider \(\tilde{\delta} \triangleq \lfloor \delta / (1 - 1/\lg n) \rfloor\) that is slightly lower than \(\delta\) to compensate for the rounding error when \(\mu \leq 1\). \(^5\) We then employ the exact suffix sums construction from the

\(^5\) If \(\tilde{\delta} = 1\), then we simply apply the exact algorithm from the previous subsection.
previous section for window size of $s \triangleq \lceil n/\nu + 1 \rceil$ (the number of chunks that can overlap with the window) over a stream of integers in $\{1, \ldots, z\}$, where $z \triangleq \lceil \mu^{-1} \nu \rceil$ is a bound on the resulting items. We use $\rho$ to denote the input that represents the current block.

The query procedure is also a bit tricky. Intuitively, we can estimate the sum of the last $i$ items by querying $A$ for the sum of the last $i/\nu$ inserted values and multiplying the result by $\delta$; but there are a few things to keep in mind. First, $i/\nu$ may not be an integer. Next, the values within the current chunk (that has not ended yet) are not recorded in $A$. Finally, we are not allowed to overestimate, so $r$'s propagation may be an issue.

To address the first issue, we weight the oldest chunk's $\rho$ value by the fraction of that chunk that is still in the window. For the second, we add the value of $r$ to the estimation, where $r$ is the sum of rounded elements. Notice that we do not reset the value of $r$ but rather propagate it between chunks. Finally, to assure that our algorithm never overestimates we subtract $\delta - 1/2$ from the result. Our algorithm uses the following variables:

- $A$ - an exact suffix sum algorithm, as described in the previous section. It allows computing suffix sums over the last $s = \lceil n/\nu + 1 \rceil$ elements on a stream of integers in $\{1, \ldots, z\}$.
- $r$ - tracks the sum of elements that is not yet recorded in $A$.
- $\rho$ - the offset within the chunk.

A pseudo code of our method appears in Algorithm 1. Next follows a memory analysis of the algorithm.

\begin{algorithm}
1: Initialization: $r \leftarrow 0, o \leftarrow 0, A.\text{init()}$
2: function $\text{ADD(element } x)$
3: \hspace{1em} $o \leftarrow (o + 1) \mod \nu$
4: \hspace{1em} $r \leftarrow r + \text{Round}_\delta(x)$
5: \hspace{2.5em} if $o = 0$ then \hfill $\triangleright$ End of a chunk
6: \hspace{3em} $\rho \leftarrow \lceil \delta^{-1} \cdot r \rceil$
7: \hspace{1em} $r \leftarrow r - \tilde{\delta} \cdot \rho$
8: \hspace{1em} $A.\text{Add}(\rho)$
9: function $\text{QUERY}(i)$
10: \hspace{1.5em} if $i \leq o$ then \hfill $\triangleright$ Queried within the current chunk
11: \hspace{2em} return $r - \left( \delta - 1/2 \right)$
12: \hspace{1.5em} else
13: \hspace{2em} $\text{numElems} \leftarrow \lceil \frac{i - o}{\nu} \rceil$
14: \hspace{2em} $\text{totalSum} \leftarrow A.\text{QUERY}(\text{numElems})$
15: \hspace{2em} $\text{oldest}_\rho \leftarrow \text{totalSum} - A.\text{QUERY}(\text{numElems} - 1)$
16: \hspace{2em} $\text{out} \leftarrow (\nu - (i - o) \mod \nu)$
17: \hspace{2em} return $r - \left( \delta - 1/2 \right) + \tilde{\delta} \cdot \text{totalSum} - \text{out}$
\end{algorithm}

Thus, we conclude that our algorithm is succinct if the error satisfies $\delta = o(\ell \cdot n/\lg n)$. We note that a $\lceil \lg n \rceil$ bits lower bound for Basic-Summing with an additive error was shown in [3], even when only fixed sized windows (where $i \equiv n$) are considered. Thus, our algorithm always requires $O(B_{\ell,n,\delta})$ space, even if $\delta = \Omega(\ell \cdot n/\lg n)$. Here, $B_{\ell,n,\delta} = \lceil n/\lceil \delta/\ell \rceil \rceil \lg \max(\lceil \ell/\delta \rceil, 1) + 1$ is the lower bound for static data shown in Theorem 9.

\begin{corollary}
Let $\ell, n, \delta \in \mathbb{N}^+$ such that $\mu \triangleq \delta/\ell$ satisfies

$$(\mu = o(n/\lg n)) \land (\lceil \mu = o(1) \rceil \lor (\mu = \omega(1)) \lor (\mu \in \mathbb{N}) \lor (\mu^{-1} \in \mathbb{N})),$$

then Algorithm 1 is succinct. For other parameters, it uses $O(B_{\ell,n,\delta})$ space.
\end{corollary}
We now state the correctness of our algorithm.

**Theorem 16 (⋆).** Algorithm 1 solves $\text{ssA}$ while processing elements and answering queries in constant time.

---

**References**

Abstract

We study multi-finger binary search trees (BSTs), a far-reaching extension of the classical BST model, with connections to the well-studied $k$-server problem. Finger search is a popular technique for speeding up BST operations when a query sequence has locality of reference. BSTs with multiple fingers can exploit more general regularities in the input. In this paper we consider the cost of serving a sequence of queries in an optimal (offline) BST with $k$ fingers, a powerful benchmark against which other algorithms can be measured.

We show that the $k$-finger optimum can be matched by a standard dynamic BST (having a single root-finger) with an $O(\log k)$ factor overhead. This result is tight for all $k$, improving the $O(k)$ factor implicit in earlier work. Furthermore, we describe new online BSTs that match this bound up to a $(\log k)^{O(1)}$ factor. Previously only the “one-finger” special case was known to hold for an online BST (Iacono, Langerman, 2016; Cole et al., 2000). Splay trees, assuming their conjectured optimality (Sleator and Tarjan, 1983), would have to match our bounds for all $k$.

Our online algorithms are randomized and combine techniques developed for the $k$-server problem with a multiplicative-weights scheme for learning tree metrics. To our knowledge, this is the first time when tools developed for the $k$-server problem are used in BSTs. As an application of our $k$-finger results, we show that BSTs can efficiently serve queries that are close to some recently accessed item. This is a (restricted) form of the unified property (Iacono, 2001) that was previously not known to hold for any BST algorithm, online or offline.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms, Theory of computation → Data structures design and analysis

Keywords and phrases binary search trees, dynamic optimality, finger search, $k$-server
Acknowledgements
We thank Nikhil Bansal and Greg Koumoutsos for insightful discussions.

1 Introduction

The binary search tree (BST) is the canonical comparison-based implementation of the dictionary data type for maintaining ordered sets. Dynamic BSTs can be re-arranged after every access via rotations and pointer moves starting from the root. Various ingenious techniques have been developed for dynamically maintaining balanced BSTs, supporting search, insert, delete, and other operations in time $O(\log n)$, where $n$ is the size of the dictionary (see e.g. [31, §6.2.2], [40, §5]).

In several applications where the access sequence has strong locality of reference, the worst-case bound is too pessimistic (e.g. in list merging, adaptive sorting, or in various geometric problems). A classical technique for exploiting locality is finger search. In finger search trees, the cost of an access is typically $O(\log d)$, where $d$ is the difference in rank between the accessed item and a finger ($d$ may be much smaller than $n$). The finger indicates the starting point of the search, and is either given by the user, or (more typically) it points to the previously accessed item. Several special purpose tree-like data structures have been designed to support finger search.

In 1983, Sleator and Tarjan [49] introduced Splay trees, a particularly simple and elegant “self-adjusting” BST algorithm. In 2000, Cole et al. [16, 15] showed that Splay matches (asymptotically) the efficiency of finger search, called in this context the dynamic finger property. This is remarkable, since Splay uses no explicit fingers; every search starts from the root. The result shows the versatility of the BST model, and has been seen as a major (and highly nontrivial) step towards “dynamic optimality”, the conjecture of Sleator and Tarjan that Splay trees are constant-competitive.

BSTs can also adapt to other kinds of locality. The working set property [49] requires the amortized cost of accessing $x$ to be $O(\log t)$, where $t$ is the number of distinct items accessed since the last access of $x$. Whereas dynamic finger captures proximity in keyspace, the working set property captures proximity in time. In 2001, Iacono [26] proposed a unified property that generalizes both kinds of proximity. Informally, a data structure with the unified property is efficient when accessing an item that is close to some recently accessed item. It is not known whether any BST data structure has the unified property.

Recently, Iacono and Langerman [28] studied the lazy finger property (Bose et al. [8]), and showed that an online algorithm called Greedy BST satisfies it. The lazy finger property requires the amortized cost of accessing $x$ to be $O(d)$, where $d$ is the distance (number of edges) from the previously accessed item to $x$ in the best static reference tree. This property is stronger than the dynamic finger property [8], and it is not known to hold for Splay.

In this paper we study a generalization of the lazy finger property; instead of a single finger stationed at the previously accessed item, we allow $k$ fingers to be moved around arbitrarily. An access is performed by moving any of the fingers to the requested item. Cost

---

4 To simplify notation, we let $\log (x)$ denote $\log_2 (\max\{2, x\})$.

5 The initial 1977 design of Guibas et al. [23] was refined and simplified by Brown and Tarjan [10] and by Huddleston and Mehlhorn [25]. Further solutions include [51, 50, 32, 30], see also the survey [9]. Randomized treaps [46] and skip lists [43] can also support finger search.

6 Greedy BST was discovered by Lucas in 1988 [37] and later independently by Munro [42]. Demaine et al. [17] transformed it into an online algorithm.
is proportional to the total distance traveled by the fingers. We assume that the fingers move according to an optimal strategy, in an optimally chosen static tree, with a priori knowledge of the entire access sequence. The cost of this optimal offline execution with \( k \) fingers is an intrinsic measure of complexity of a query sequence, and at the same time a benchmark that algorithms in the classical model can attempt to match. Parameter \( k \) describes the strength of the bound: the case \( k = 1 \) is the lazy finger, at the other extreme, at \( k = n \), each item may have its own finger, and all accesses are essentially free.

Our main result is a family of new online\(^7\) dynamic BST algorithms (in the standard model, where every access starts at the root), matching the \( k \)-finger optimum on sufficiently long sequences, up to an overhead factor with moderate dependence on \( k \) and no dependence on the dictionary size or on the number of accesses in the sequence.

Our online BST combines three distinct techniques: (1) an offline, one-finger BST simulation of a multi-finger execution (the technique is a refinement of an earlier construction \cite{18}), (2) online \( k \)-server algorithms that can simulate the offline optimal multi-finger strategy, and (3) a multiplicative-weights scheme for learning a tree metric in an online fashion.

The fact that “vanilla” BSTs can, with a low overhead, simulate a much more powerful computational model further indicates the strength and versatility of the BST model. As an application, we show that our online BST algorithms satisfy a restricted form of the unified property; previously no (online or offline) BST was known to satisfy such a property.

If there is a constant-competitive BST algorithm, then it must match our \( k \)-finger bounds. The two most promising candidates, Splay and Greedy BST (see e.g. \cite{27}) were only shown (with considerable difficulty) to satisfy variants of the one-finger, i.e. lazy finger property. To obtain our online BSTs competitive for other values of \( k \), we combine sophisticated tools developed for other online problems, as well as our refinement of a previous (highly nontrivial) construction for simulating multiple fingers. These facts together may hint at the formidable difficulty (more pessimistically: the low likelihood) of attaining dynamic optimality by simple and natural BST algorithms such as Splay or Greedy.

**BST and finger models. Main results.** Now, we introduce the formal statements of our results. In the dynamic BST model a sequence of keys is accessed in a binary search tree (BST), and after each access, the tree can be reconfigured via a sequence of rotations and pointer moves starting from the root. (There exist several alternative but essentially equivalent models, see \cite{52, 17}.) Denote the space of keys (or elements) by \([n]\). For a sequence \( X = (x_1, \ldots, x_m) \in [n]^m \), denote by \( \text{OPT}(X) \) the cost of the optimal offline BST for accessing \( X \).\(^8\) Arguably the most important question in the BST model is the dynamic optimality conjecture, i.e. the existence of an online BST whose cost is \( O(\text{OPT}(X)) \) for every \( X \).

A BST optimality property is an inequality between \( \text{OPT}(X) \) and some function \( f(X) \), that holds in the BST model. (If \( \text{OPT}(X) \leq f(X) \) for all \( X \) is a BST optimality property, then every \( O(1) \)-competitive algorithm must cost at most \( O(f(X)) \).)

Several natural BST properties have been suggested over the last few decades. For instance, the static finger property \cite{49} states \( \text{OPT}(X) = O(\text{SF}(X)) \), for \( \text{SF}(X) = \sum_j \log |x_i - j| \), where \( j \in [n] \) is a fixed element (finger). The static optimality property \cite{49} is \( \text{OPT}(X) = O(\text{SO}(X)) \), where \( \text{SO}(X) = \min_R \sum_i d_R(x_i) \). Here \( R \) is a static BST, and \( d_R(x) \) is the depth of \( x \) in \( R \).

\(^{7}\) An online BST algorithm can base its decisions only on the current and past accesses. An offline algorithm knows the entire access sequence in advance.

\(^{8}\) To avoid technicalities, we only consider access (i.e. successful search) operations and assume \( m \geq n \).
For the dynamic finger property [49], $DF(X) = \sum_{t} \log |x_t - x_{t+1}|$, and for working set [49], $WS(X) = \sum_{t} \log \rho_t(x_t)$, where $\rho_t(a)$ is the number of distinct keys accessed between time $t$ and the last time at which $a$ was accessed (all keys assumed accessed at time zero).

In 2001, Iacono [26] initiated the study of a property that would “unify” the latter two notions of efficiency and exhibited a data structure (not a BST) achieving this property. This unified bound is defined as $UB(X) = \sum_{t} \min_{t' < t} \log \left( |x_t - x_{t'}| + \rho_t(x_{t'}) \right)$. Dynamic finger and working set are in general, not comparable. On the other hand, $UB(X) \leq WS(X)$ clearly hold, justifying the name of the unified bound.

Despite several attempts, the question whether the unified bound is a valid BST property remains unclear; it was shown in [20] that $OPT(X) = O(UB(X) + m \log \log n)$, and in [11, 26] that the unified bound is valid in some other (non-BST) models9.

We show that a unified bound with “bounded time-window” holds in the BST model:

**Theorem 1.** For every integer $\ell \geq 1$, every sequence $X$ and some fixed function $\beta(\cdot)$,

$$OPT(X) \leq \beta(\ell) \cdot UB^\ell,$$

where $UB^\ell = \sum_{t} \min_{t' \in [t-\ell, t]} \log \left( |x_t - x_{t'}| + \rho_t(x_{t'}) \right)$.

Observe that $UB(X) = UB^m(X) \leq \cdots \leq UB^1(X) = DF(X)$. Prior to our work it was not known whether the theorem holds when $\ell = 2$, i.e. no known BST property subsumes this property even when $\ell = 2$. Thus, Theorem 1 establishes the first BST property that combines the efficiencies of time- and keyspace-proximity without an additive term.10

Recently Bose et al. [8] introduced the lazy finger property, $LF(X) = \min_{R} \sum d_R(x_t, x_{t+1})$. Here distance is measured in a static reference BST $R$, optimally chosen for the entire sequence. The lazy finger bound can be visualized as follows: accesses are performed in the reference tree by moving a unique finger from the previously accessed item to the requested item. The lazy finger property is rather strong: Bose et al. show that it implies the dynamic finger and static optimality properties, which in turn imply static finger.

Our main tool in proving Theorem 1 is a generalization of the lazy finger property allowing multiple fingers. The model is motivated by the famous $k$-server problem. For an input sequence $X \in [n]^m$ and a static BST $R$ with nodes associated with the keys in $[n]$, we have $k$ servers located initially at arbitrary nodes in $R$. At time $t = 1, \ldots, m$, the request $x_t$ arrives, and we move a server of our choice to the node of $R$ that stores $x_t$. The cost for serving a sequence $X$ is equal to the total movement in $R$ to serve the sequence $X$.

Denote by $F_R^k(X)$ the cost of the optimal (offline) strategy that serves sequence $X$ in $R$ with $k$ servers, minimized over all possible initial server locations. Let $F^k(X) = \min_{R} F_R^k(X)$. We call $F^k(X)$ the $k$-finger cost of $X$. We remark that the value of $F_R^k(X)$ is polynomial-time computable for each $R$, $k \in \mathbb{N}$, and $X \in [n]^m$ by dynamic programming. Clearly, $F^1(X) \geq F^2(X) \geq \cdots \geq F^m(X)$ holds for all $X$.

We first show that one can simulate any $k$-finger strategy in the BST model, in a near-optimal manner. In particular, we prove the following tight result.

**Theorem 2.** $OPT(X) \leq O(\log k) \cdot F^k(X)$.

The proof of Theorem 2 is a refinement of an earlier argument [18], improving the overhead factor from $O(k)$ to $O(\log k)$. The logarithmic dependence on $k$ is, in general, the best possible. To see this, consider a sequence $S$ of length $m$, over $k$ distinct items with average cost $\Omega(\log k)$ (e.g. a random sequence from $[k]^m$ does the job). While $OPT(S) = \Theta(m \log k)$, clearly $F^k(X) = O(m)$, as each of the $k$ items can be served with its own private finger.

---

9 Another attempt to study the bounds related to the unified bound was done in [24].
10 The proof of Theorem 1 implies in fact a stronger, weighted form, which we omit for ease of presentation.
In the definition of $F^k(X)$ we assume a static reference tree $R$ for the $k$-finger execution. The offline BST simulation in the proof of Theorem 2 works in fact (with the same overhead) even if $R$ is dynamic, i.e. if the multi-finger adversary can perform rotations at any of the fingers. In this case, however, the $k$-finger bound is too strong to be useful; already the $k = 1$ case captures the dynamic BST optimum. Our next result is the online counterpart of Theorem 2. In this case, the restriction that $R$ is static is essential.

**Theorem 3.** There exists an online randomized BST algorithm whose cost for serving $X \in [n]^m$, is $O((\log k)^7) \cdot F^k(X) + \rho(n)$, for some fixed function $\rho(\cdot)$.

The result can be interpreted as follows. On sufficiently long access sequences, there is an online BST algorithm (in fact, a family of them) competitive with the $k$-finger bound, up to an overhead factor with moderate dependence on $k$. The randomized algorithm (as is standard in the online setting) assumes an oblivious adversary that does not know in advance the outcomes of the algorithm’s random coin-flips. The use of randomness seems essential to our approach. We propose as intriguing open questions to find a deterministic online BST with comparable guarantees and to narrow the gap between the online and offline results.

Due to its substantial amount of computation (outside the BST model), our online algorithm is of theoretical interest only. Nonetheless, the connection with the $k$-server problem allows us to “import” several techniques to the BST problem; some of these, such as the double coverage heuristic for $k$-server [14] are remarkably simple and may find their way to practical BST algorithms.

The strength of the $k$-finger model lies in the $k$-server abstraction. In order to establish a BST property of the form $\text{OPT}(X) \leq \beta(\ell) \cdot O(g(X))$, it is now sufficient to prove $F^1(X) \leq (\beta(\ell)/\log \ell) \cdot O(g(X))$. In other words, our technique reduces the task of bounding the cost in the BST model to designing $k$-server strategies, which typically admits much cleaner combinatorial arguments. We illustrate this approach by showing that the unified property with a fixed time-window holds in the BST model.

**Theorem 4.** For some fixed functions $\alpha(\cdot), \gamma(\cdot)$, we have: $F^{\alpha(\ell)}(X) \leq \gamma(\ell) \cdot UB^\ell$.

Theorems 4 and 2 together imply Theorem 1. Moreover, Theorem 3 implies that the property holds for online BST algorithms (we later specify the involved functions).

The $k$-finger approach can be used to show further BST properties. For example, we connect decomposability (refer to §4 for definitions) and finger properties by showing that even one finger is enough to obtain the traversal property in significantly generalized form.

**Theorem 5.** Let $X$ be a $d$-decomposable sequence. Then $F^1(X) = O(\log d) \cdot |X|$.

As a corollary, using the recent result by Iacono and Langermann [28], we resolve an open problem in [13], showing that Greedy costs at most $O(\log d) \cdot |X|$ on every $d$-decomposable sequence, matching the lower bound in [13].

In another direction, we connect multiple fingers and generalized monotone sequences. In [13], we showed that $\text{OPT}(X) \leq |X| \cdot 2^{O(d^2)}$ on every $d$-monotone sequence $X$; a sequence is $d$-monotone if it can be decomposed into $d$ increasing or $d$ decreasing sequences. Using the $k$-finger technique, we show the stronger BST property $\text{OPT}(X) \leq O(d \log d) \cdot |X|$.

Concerning simple and natural BST algorithms (Splay and Greedy), we give evidence that the strongest results in the literature may still be far from settling the dynamic optimality conjecture. To this end, we describe a class of sequences for which increasing the number of fingers by one can create an $\Omega(\log n)$ gap. More precisely, we show the following:

---

11 Independently of our work, Goyal and Gupta [22] showed the same result using a charging argument.
Theorem 6. For every integer $k$, there is a sequence $S_k$ such that $F^{k-1}(S_k) = \Omega\left(\frac{n}{k} \log(n/k)\right)$ but $F^k(S_k) = O(n)$.

Theorem 6 shows that the multi-finger bounds form a fine-grained hierarchy. For small $k$, our online algorithm (Theorem 3) can match these bounds (up to a constant factor). However, any online BST (such as Splay or Greedy) must also match the dependence of $O(\log k)$ in the upper bound of $O(\log k) \cdot F^k(X)$, in order to be constant-competitive.

Techniques. The $k$-server problem. The $k$-server problem, introduced by Manasse, McGeoch, and Sleator [38] in 1988 is a central problem in online algorithms: Is there an online deterministic strategy for serving a sequence of requests by moving $k$ servers around, with a total movement cost at most $k$ times the optimal offline strategy? The question in its original form, for arbitrary metric spaces, remains open. Nonetheless, the problem has inspired a wealth of results and a rich set of techniques, many of which have found applications outside the $k$-server problem. A full survey is out of our scope, we refer instead to some prominent results [21, 34, 44, 3, 2], and the surveys [6, §10, §11], [33]. Most relevantly for us, Chrobak and Larmore [14] gave in 1991, an intuitive, deterministic, $k$-competitive algorithm for tree metrics, and the very recently announced breakthrough of Lee [35], building on Bubeck et al. [12], gives an $O((\log k)^6)$-competitive randomized algorithm for arbitrary metrics.

Our online BST algorithm relies on an online $k$-server in an almost black box fashion (the metric space underlying the $k$-server instance is induced by a static reference BST). Thus, improvements for $k$-server would directly yield improvements in our bounds. Despite the depth and generality of $k$-server (e.g. it also models the caching/paging problem), to our knowledge it has previously not been related to the BST problem.\footnote{In his work on a generalized $k$-server problem, Sitters [48] asks whether the work-function (WF) technique [34] for $k$-server may have relevance for BSTs. Indeed, we can use WF as an $O(k)$-competitive component of our online BSTs, but for our special case of tree-metrics, the technique of [14] is much simpler. Whether WF may be used (in different ways) to obtain competitive BSTs remains open.}

It is known that in an arbitrary metric space with at least $k + 1$ points, no deterministic online algorithm may have a competitive ratio better than $k$. In the randomized case the lower bound $\Omega(\log k / \log \log k)$ holds, see e.g. [33]. (The lower bounds thus apply for a metric induced by a BST, for all $k < n$.) These results imply a remarkable separation between the $k$-server and BST problems. Dynamic optimality would require, by Theorem 2, a BST cost of $O(\log k) \cdot F^k$. To match this, an online BST may not implicitly perform a deterministic $k$-server execution, since, in that case its overhead would have to be $\Omega(k)$. This indicates that improving Theorem 3 will likely require tools significantly different from $k$-server, which is surprising, given the similarity of the two formulations.

Our online BST learns the metric induced by the optimal reference tree using a multiplicative weights update (MWU) scheme. The technique has a rich history, and a recent emergence as a powerful algorithmic tool (we refer to the survey of Arora, Hazan, and Kale [1]). MWU or closely related techniques have been used previously in data structures (including for BST-related questions), see e.g. [5, 4, 27, 29]. Specifically, Iacono [27] obtains, using MWU, an online BST that is constant-competitive on sufficiently long sequences, if any online BST is constant-competitive. As we relate online BSTs with an offline strategy, the results are not directly comparable.
Further open questions and structure of the paper. The main open question raised by our work is whether natural algorithms such as Splay or Greedy match the properties of our new BST algorithms. (This must be the case, if Splay and Greedy are, as conjectured, $O(1)$-competitive). We suggest the following easier questions. Do Splay or Greedy satisfy the unified bound with a time-window of 2 steps? Does Splay satisfy the lazy finger or the 2-monotone bounds? Does Greedy satisfy the 2-server bound?

Except for Theorems 2 and 5, the factors in our results are not known to be tight. Improving them may reveal new insight about the power and limitations of the BST model.

In §2 we describe our offline BST simulation. In §3 we describe our new family of online algorithms. In §4 we prove the main applications and further observations.

2 Offline simulation of multi-finger BSTs (Theorem 2)

Let $k \in \mathbb{N}$, let $T$ be a BST on $[n]$, and let $X = (x_1,\ldots,x_m) \in [n]^m$ be an access sequence. A $k$-finger strategy consists of a sequence $\bar{f} \in [k]^m$ where $f_i \in [k]$ specifies the finger that serves access $x_i$. Let $\bar{\ell} \in [n]^k$ be the initial vector, where $\ell_i \in [n]$ gives the initial location of finger $i$. The cost of strategy $(\bar{f}, \bar{\ell})$ is $F^k_{T, \bar{f}, \bar{\ell}}(X) = \sum_{i=1}^{m} (1 + d_T(x_i, x_{\ell(f_i,t_i)})$ where $\sigma(i, t) = \max\{j < t \mid f_j = i\}$ is the location of finger $i$ before time $t$, and $\sigma(i, 1) = \ell_i$. Let $F^k_T(X) = \min_{\bar{f},\bar{\ell}} F^k_{T, \bar{f}, \bar{\ell}}(X)$. In other words, for a fixed BST $T$ on keyset $[n]$, $F^k_T(X)$ is the $k$-server optimum for serving $X$ in the metric space of the tree $T$. (Note that the tree is unweighted, and the distance $d_T(\cdot, \cdot)$ counts the number of edges between two nodes in $T$.) We define $F^k(X) = \min_T F^k_T(X)$. It is clear from the definition that $F^1(X) \geq F^2(X) \geq \cdots \geq F^n(X) = m$ for all $X$.

Observe that we implicitly assume that during every access at most one server moves. In addition, we may assume that if some server is already placed at the requested node, then no movement happens. Algorithms with these two restrictions are called lazy. As argued in the $k$-server literature (see e.g. [33]), non-lazy server movements can always be postponed to a later time, keeping track of the “virtual” locations of servers. In other words, every $k$-server algorithm can be simulated by a lazy algorithm, without additional cost. We therefore assume throughout the paper that $k$-server/$k$-finger executions are lazy.

Consider some (lazy) $k$-finger execution $(\bar{f}, \bar{\ell})$ in tree $T$, for access sequence $X$. We can view $\bar{f}$ as an explicit sequence of elementary steps $S = S^k_{\bar{f}, \bar{\ell}}$ where in each step we move one of the fingers to its parent or to one of its children in $T$. We further allow $S$ to contain rotations at a finger in $T$ (although $k$-finger strategies as described above do not generate rotations). The position of a finger is maintained during a rotation.

We show how $S$ can be simulated in a standard dynamic BST. If in $S$ a finger visits a node, then the (single) pointer in the BST also visits the corresponding node, therefore all accesses are correctly served in the BST. Every elementary step in $S$ is mapped to (amortized) $O(\log k)$ elementary steps (pointer moves and rotations) in the BST. This immediately implies Theorem 2, since, if we can simulate an arbitrary $k$-finger execution, then indeed we can simulate the optimal $k$-finger execution on the best static tree. Assuming that the initial conditions $T$ and $\bar{\ell}$ are known, the steps of $S$ are simulated one-by-one, without any lookahead. Thus, insofar as the $k$-finger execution is online, the BST execution is also online (this fact is used in §3).

Let us describe simulation by a standard BST $T'$ of a $k$-finger execution $S$ in a BST $T$. The construction is a refinement of the one given by Demaine et al. [18], see also [19]. (We improve the overhead factor from $O(k)$ to $O(\log k)$.) The main ingredients are: (1) Making sure that each item with a finger on it in $T$ has depth at most $O(\log k)$ in $T'$. (In [18], each
finger may have depth up to \( O(k) \) in \( T' \).) (2) Implementing a deque data structure within \( T' \) so that each finger in \( T \) can move to any of its neighbors, or perform a rotation, with cost \( O(\log k) \) amortized. (In [18], this cost is \( O(1) \) amortized.)

Given these ingredients, to move a finger \( f \) to its neighbor \( x \) in \( T \), we can simply access \( f \) from the root of \( T' \) in \( O(\log k) \) steps, and then move \( f \) to \( x \) in \( T' \) in \( O(\log k) \) amortized steps, with a similar approach for a rotation at \( f \). Hence, the overhead factor is \( O(\log k) \). We sketch the main technical ideas, postponing the details to Appendix A.

Consider the tree \( S \) induced by the current fingers and the paths connecting them in \( T \). The tree \( S \) consists of finger-nodes and non-finger nodes of degree 3 (both types of nodes are called pseudo-fingers), and paths of non-finger nodes of degree 2 connecting pseudo-fingers with each other, called tendons. Tendons can be compressed into a BST structure that allows their traversal between the two endpoints in \( O(1) \) steps.

We maintain \( S \) as a root-containing subtree of our BST \( T' \), called the hand. Due to the compression of the tendons, the relevant part of \( S \) has size \( O(k) \). The description so far, including the terminology, is identical to the one in [18, §2]. Our construction differs in the fact that it maintains the hand, i.e. the compressed representation of \( S \) as a balanced BST. This guarantees the reachability of fingers in \( O(\log k) \) instead of \( O(k) \) steps, i.e. property (1).

When a finger in \( T \) moves or performs a rotation, the designation of some (pseudo)finger, or tendon nodes may change. Such changes can be viewed as the insertion or deletion of items in the tendons. As these operations happen only at certain places within the tendons, they can be implemented efficiently. We implement tendons with the same BST-based deque as [18]. The construction appears to be folklore, we describe it in Appendix A.1 for completeness.

We depart again from [18], as the operation affecting the (pseudo)finger and tendon nodes can trigger a re-balancing of the hand, which may again require \( O(\log k) \) operations to fix, i.e. property (2). Any efficient balancing strategy (e.g. red-black tree) may be used.

## 3 Online simulation of multi-finger BSTs (Theorem 3)

Consider the optimal (offline) \( k \)-finger execution \( \vec{f} \) for access sequence \( X \in [n]^m \), with static reference tree \( T \) and initial finger-placement \( \vec{\ell} \). We wish to simulate it by a dynamic online BST. The construction proceeds in two stages: (1) A simulation of \( \vec{f} \) by a sequence \( S \) of steps that describe finger-movements and rotations-at-fingers, starting from an arbitrary BST \( T_0 \) and arbitrary finger locations \( \vec{\ell}_0 \). The sequence \( S \) is online, i.e. it is constructed without knowledge of the optimal initial state \( T, \vec{\ell} \), and it correctly serves the sequence \( X \), as its elements are revealed one-by-one. (2) A step-by-step simulation of \( S \) by a standard BST algorithm using the result of §2. Since \( S \) is online, the BST algorithm is also online.

As before, we denote by \( F^k(X) = F^k_{T, T', \vec{f}}(X) \) the cost of the optimal offline execution. Observe that this is exactly the \( k \)-server optimum with the tree metric defined by \( T \) and initial configuration of servers \( \vec{\ell} \). If \( T \) and \( \vec{\ell} \) were known, we could conclude part (1) by running an arbitrary online \( k \)-server algorithm defined on tree metrics.

To this end, we mention two online \( k \)-server algorithms, the deterministic “double coverage” algorithm of Chrobak and Larmore [14] (Algorithm A) and the very recently announced randomized algorithm of Lee [35, 12] (Algorithm B). It is known that the cost of Algorithms A, resp. B is at most \( k \)-times, resp. \( O((\log k)^6) \) times \( F^k \). We only describe Algorithm A, as it is particularly intuitive. To obtain the claimed result, we need the much more complex Algorithm B. (By using Algorithm A we get an overall factor \( O(k \log k) \).)
During the execution of Algorithm A, given a current access request $x_t$, call those servers (fingers) active, whose path to $x_t$ in $T$ does not contain another server. If several servers are in the same location, one of them is chosen arbitrarily to be active. Algorithm A serves $x_t$ as follows: as long as there is no server on $x_t$, move all active servers one step closer to $x_t$. Observe that as servers move, some of them may become inactive. Algorithm A (as described) may need to move multiple servers during one access. It can, however, easily be transformed into a lazy algorithm, as discussed in §2.

Remains the issue that the optimal initial $T$ and $\tilde{T}$ are not known. Let $B_1, \ldots, B_N$ be instances of an online $k$-server algorithm (in our case Algorithm B), one for each combination of initial tree $T$ and initial server-placement $\tilde{T}$. Note that $N = O(4^n \cdot n^k)$. Let $M$ be a “meta-algorithm” that simulates all $B_j$’s for $j = 1, \ldots, N$, competitive on sufficiently long input with the best $B_j$. Algorithm $M$ processes $X$ in epochs of length $M = n \log n$, executing in the $i$-th epoch, for $i = 1, \ldots, \lceil m/M \rceil$, some $B_{\tau(i)}$ according to a (randomized) choice $\tau(i)$.

Suppose that $\tilde{T}$ and $T^*$ describe the state of $B_{\tau(i)}$ chosen by $M$ at the beginning of the $i$-th epoch. To switch to the state $\tilde{T}$, $T^*$, $M$ takes $O(n \log n)$ elementary steps: (1) rotate the current tree to a balanced tree using any of the fingers ($O(n)$ steps), (2) move all fingers to their location in $\tilde{T}$ ($k$ times $O(\log n)$ steps), (3) use an arbitrary finger $f$ to rotate the tree to $T^*$ ($O(n)$ steps), (4) move $f$ back to its location in $\tilde{T}$ ($O(n)$ steps). Since $M = n \log n$, the cost of switching can be amortized over the epoch.

The choice of $B_{\tau(i)}$ for epoch $i$ is done according to the multiplicative-weights (MW) technique [1], based on the past performance of the various algorithms. Our experts are the online executions $B_1, \ldots, B_N$, our $i$-th event is the portion of $X$ revealed in the $i$-th epoch, the loss of the $j$-th expert for the $i$-th event is the cost of $B_j$ in the $i$-th epoch. Let $C_{max}$ denote the maximum possible loss of an expert for an event (we may assume $C_{max} \leq n \cdot M$).

It follows from the standard MW-bounds [1, Thm. 2.1], that for an arbitrary $\epsilon \in (0, 1)$, the cost of $M$ on $X$ is at most $\min_j (1 + \epsilon)C_j + \frac{C_{max}}{\epsilon} \ln N$, where $C_j$ is the cost of expert $B_j$ for the entire $X$; in particular, $B_j$ may correspond to the optimal offline choice $\tilde{T}, T$, in which case $C_j = O((\log k)^6 \cdot F^k(X))$.

Thus, for e.g. $\epsilon = 1/2$, we obtain that the cost of $M$ on $X$ is at most $O((\log k)^6 \cdot F^k(X) + O(n^3 \log^2 n))$. The output of $M$ is an online sequence $S_M$ of rotations and finger moves, starting from an arbitrary initial state $T_0$ and $\tilde{T}_0$. Note that while $M$ needs to evaluate the costs and current states for all experts in all epochs (an extraordinary amount of computation), only one of the experts interacts with the tree at any time. Thus, $S_M$ is a standard sequence of steps which can be simulated by a standard BST algorithm according to Theorem 2, at the cost of a further $O(\log k)$ factor. This concludes the proof of Theorem 3.

4 Applications of the multi-finger property

In this section we show that every BST algorithm that satisfies the $k$-finger property also satisfies the unified bound with fixed time-window (Application 1), is efficient on decomposable sequences (Application 2), and on generalized monotone sequences (Application 3).

Application 1. Combined space-time sensitivity (Theorem 4). Recall the definition of $UB^\ell$ in Theorem 1 for a sequence $X = (x_1, \ldots, x_m) \in \llbracket n \rrbracket^m$. We connect this quantity with the $k$-finger cost, from which Theorem 4 immediately follows.

Theorem 7. For every $\ell$, $F(0)(X) = O(\ell) \cdot UB^\ell(X)$. 

ISAAC 2018
Multi-Finger Binary Search Trees

Since we are only concerned with the case when \( \ell \) is constant, we may drop the term \( \rho_t(x_t) \) in the definition of UB\(^\ell\) (whose value is always between 1 and \( \ell \)).

We prove Theorem 7 via another bound in which distances are measured in a static reference BST: \( \ell\)-DistTree\(_T\)(\( X \)) = \( \sum_{i=1}^{m} \min_{1 \leq j < i} \{d_T(x_i, x_j) + 1\} \).  

**Lemma 8.** \( \min_T \ell\)-DistTree\(_T\)(\( X \)) = \( O(\text{UB}^\ell(\( X \))) \).

**Proof.** By [46, Thm. 4.7], there is a randomized BST \( \tilde{T} \) such that the expected distance between elements \( i \) and \( j \) is \( E[d_{\tilde{T}}(i, j)] = \Theta(\log |i - j|) \). Therefore,

\[
\min_T \ell\text{-DistTree}_T(\( X \)) \leq E[\sum_{i=1}^{m} \min_{1 \leq j < i} \{d_T(x_i, x_j) + 1\}] = \sum_{i=1}^{m} E[\min_{1 \leq j < i} \{d_T(x_i, x_j) + 1\}]
\]

\[
\leq \sum_{i=1}^{m} \min_{1 \leq j < i} \{E[d_T(x_i, x_j) + 1]\} = \sum_{i=1}^{m} \min_{1 \leq j < i} \{O(\log |x_i - x_j|)\} = O(\text{UB}^\ell(\( X \))).
\]

It is now sufficient to show that \( F_{\ell}^{(\ell)}(\( X \)) = O(\ell!) \cdot \ell\text{-DistTree}_T(\( X \)), \) for all \( X \) and \( T \), i.e. to describe an \( (\ell!)\)-finger strategy in \( T \) for serving \( X \) with the given cost.

At a high level, our strategy is the following: (1) Define a virtual tree \( T(\( X \)) \) whose nodes are the requests \( x_i \) for \( i = 1, \ldots, m \). The virtual tree captures the proximities between the requests, with each \( x_i \) having as parent the nearest request \( x_j \) within a fixed time-window before time \( i \). Edges in \( T(\( X \)) \) are given as weights the distances between requests in \( T \). Note that the virtual tree is not necessarily binary. (2) Define a recursive structural decomposition of the tree \( T(\( X \)) \), with the property that certain blocks of this decomposition contain requests in non-overlapping time-intervals. (3) Describe a multi-finger strategy on \( T(\( X \)) \) for serving the requests, which induces a multi-finger strategy on \( T \) with the required cost. (The strategy takes advantage of the decomposition in (2).)

We describe the steps more precisely, deferring some details to Appendix B.

**The virtual tree.** Given a number \( \ell \), \( X \in [n]^m \), and a BST \( T \) over \( [n] \) with root \( r \), the virtual tree \( T = T(\ell, T, X) \) is a rooted tree with vertex-set \( \{(i, x_i) \mid i \in [n]\} \cup \{(0, x_0)\} \), where \( x_0 = r \) is the root of \( T \) and \( (0, x_0) \) is the root of \( T \). The parent of a non-root vertex \( (i, x_i) \) in \( T \) is \( (j, x_j) = \arg \min_{(i', x_i') \in [\ell - \ell]} \{d_T(x_i, x_j)\} \). In words, \( (j, x_j) \) is the root at most \( \ell \) steps before \( (i, x_i) \), closest to \( x_i \) (in \( T \)).

For each edge \( e = ((j, x_j), (i, x_i)) \), we define the weight \( w_T(e) = d_T(x_i, x_j) + 1 \). For each subtree \( H \) of \( T \), let \( w_T(H) \) be the total weight of its edges. Observe that \( w_T(T) = \ell\text{-DistTree}_T(\( X \)) \).

**Structure and decomposition of the virtual tree.** We say that a vertex \( (i, x_i) \) is before (or earlier than) \( (j, x_j) \) if \( i < j \), otherwise it is after (or later than). For every subtree \( H \) of \( T \) we denote the earliest vertex in \( H \) as \( \text{start}(H) \) and the latest vertex in \( H \) as \( \text{end}(H) \). The time-span of \( H \), denoted \( \text{span}(H) \), is \( (t_1, t_2) \) where \( (t_1, x_{t_1}) = \text{start}(H) \) and \( (t_2, x_{t_2}) = \text{end}(H) \), and \( H \) is active at time \( t \) if \( t \in \text{span}(H) \).

We describe a procedure to decompose \( T(\ell, T, X) \) into directed paths (for the purpose of analysis), defining the key notions of \( i\)-body and \( i\)-core. The procedure is called on a subtree \( H \) of \( T \), and the top-level call is \( \text{decompose}(T, \ell) \).

---

13 We let \( x_0 \) denote the root of \( T \), and distances involving negative indices are defined to be \(+\infty\).
procedure decompose($H, i$):
1. If $H$ has no edges, return.
2. Let $C(H)$ be the path from start($H$) to end($H$).
3. Call $C(H)$ an $i$-core of $H$, and call $H$ the $i$-body of $C(H)$.
4. For each connected component $H'$ in $H \setminus C(H)$ invoke decompose($H', i - 1$).

Observe that $T$ itself is an $\ell$-body. Each $i$-body $H$ consists of its $i$-core $C(H)$ and a set of $(i - 1)$-bodies that are connected components in $H \setminus C(H)$. For each of those $(i - 1)$-bodies $H'$, we say that $H$ is a parent of $H'$, defining a tree-structure over bodies. Observe that the number of ancestor bodies of an $i$-body (excluding itself) is $\ell - i$. We make a sequence of further structural observations about the virtual tree and its decomposition.

Lemma 9 (B.1).
(i) At every time $t$, there are at most $\ell$ active edges in $T(t, T, X)$.
(ii) The $i$-cores of the decomposition, for $1 \leq i \leq \ell$, partition the vertices of $T$.
(iii) Let $H$ be an $i$-body. At any time during the time-span of $H$, among the $(i - 1)$-bodies with parent $H$ at most $i - 1$ are active.
(iv) Let $H$ be an $i$-body. The $(i - 1)$-bodies with parent $H$ can be partitioned into $(i - 1)$ groups $H_1, \ldots, H_{i - 1}$ such that, for $1 \leq j \leq i - 1$ and $H', H'' \in H_j$, the time-spans of $H'$ and $H''$ are disjoint.

The strategy for moving fingers. For two vertices $(i, x_i)$ and $(j, x_j)$ in the virtual tree $T = T(\ell, T, X)$, moving a finger $f$ from $(i, x_i)$ to $(j, x_j)$ means the following: let $P = ((i_1, x_{i_1}), \ldots, (i_k, x_{i_k}))$ be the unique path from $(i, x_i) = (i_1, x_{i_1})$ to $(j, x_j) = (i_k, x_{i_k})$ in $T$. For $j = 1, \ldots, k - 1$, we iteratively move a finger $f$ from $x_{i_j}$ to $x_{i_{j+1}}$ using $d_T(x_{i_j}, x_{i_{j+1}})$ steps. Hence, the total number of steps is at most $w_T(P)$.

By serving an access in an $i$-body $H$, we mean that, for each $(j, x_j) \in V(H)$, at time $j$ there is a finger move to $x_j$ in $T$. For each $i \leq \ell$, let $nf(i)$ be the number of fingers used for serving accesses in an $i$-body. We define $nf(1) = 1$ and $nf(i) = 1 + (i - 1) \cdot nf(i - 1)$, thus, by induction, $nf(i) \leq i!$ for all $i \leq \ell$.

We now describe the strategy for moving fingers. Let $F$ be a set of fingers where $|F| = nf(\ell)$. At the beginning all fingers are at $(0, x_0)$. (In the reference tree $T$, all fingers are initially at the root $x_0$.) For $1 \leq j \leq m$, we call access($T, F, (j, x_j)$), defined below for an $i$-body $H$, set of fingers $F$, and $u \in V(H)$.

procedure access($H, F, u$):
Let $C = C(H)$ be the $i$-core of $H$, with $C = \{u_1, \ldots, u_k\}$, where $u_k$ is before $u_{k+1}$ for each $k$. For $1 \leq j \leq i - 1$, let $H_j$ be the $j$-th group of the $(i - 1)$-bodies with parent $H$ ($H_j$ defined in Lemma 9(iv)). The $i$-bodies in $H_j$ are ordered by their time-span. That is, suppose $H_j = \{H'_1, \ldots, H'_\ell\}$. For each $\ell$, if $\text{span}(H'_\ell) = (a_1, a_2)$ and $\text{span}(H'_{\ell+1}) = (b_1, b_2)$, then $a_2 \leq b_1$. Fingers in $F$ are divided into $i$ groups $F_1, \ldots, F_{i-1}, \{f_i\}$, where $|F_j| = nf(i - 1)$, for $j \leq i - 1$, and $f_i$ is a single finger.
1. If $u \in C$, then move $f_i$ to $u$ from the predecessor node of $u$ in $C$. If $u = \text{end}(H)$, then move $F$ from end($H$) to start($H$).
2. Else let $u \in V(H') \setminus V(C)$ where $H' \in H_j$. If $u = \text{start}(H')$ and $H'$ is the first $(i - 1)$-body in $H_j$, move $F_j$ from start($H$) to start($H'$). Perform access($H', F_j, u$). If $u = \text{end}(H')$ and if $H'$ is the last in $H_j$ then move $F_j$ from start($H'$) to end($H$). Otherwise, if $u = \text{end}(H')$ and there is a next $(i - 1)$-body $H''$ in $H_j$, then move $F_j$ from start($H'$) to start($H''$).
In order to give the reader more intuition, we give an alternative description. A 1-body
H consists only of its 1-core C(H). We use one finger and move it through C(H). For i > 1,
an i-body H decomposes in its i-core C(H) and i − 1 groups H1 to Hn−1 of (i − 1)-bodies.
Initially, we have nf(i) fingers on start(H). We use one finger to move down the i-core. We use
a group Fj of nf(i − 1) fingers for the j-group Hj. Let H1, . . . , Hi be the (i − 1)-cores in
Hj. We first move Fj to start(H1). Then we use the strategy recursively to move Fj through
H1. Once the group of fingers has reached end(H1), we move them to start(H2), and so on.
Once the fingers have reached end(Hi), we move them back to start(H). We coordinate (this
is not really necessary) the movement of the fingers by the order of the accesses in the access
sequence X.

From the description of access it is clear that all accesses in T are served and that nf(ℓ)
fingers are sufficient. It remains to bound the total number of steps all fingers move. For an
i-body H, let cost(H) be the total cost of calling access(H, F, u) for all u ∈ H. Let H denote the
set of (i − 1)-bodies with parent H. Let C+(H) denote the i-core C(H) augmented with the
edges connecting C(H) to the (i − 1)-bodies in H. Then:

Lemma 10 (B.2). cost(H) ≤ 2 · nf(i) · wT (C+(H)) + ∑H′∈H cost(H′).

By induction, we obtain cost(H) ≤ 2 · il · wT (H). (For i = 1 we have H = C(H).)
Since nf(ℓ) ≤ ℓ!, we have that Fℓ(T)(X) ≤ FℓT(X) ≤ cost(T). By the previous claim we
have cost(T) ≤ 2 · ℓ! · wT (T) = 2 · ℓ! · ℓ-DistTreeT(X), concluding the proof.

Application 2. Decomposable sequences (Theorem 5). Let σ = (σ(1), . . . , σ(n)) be a
permutation. For a, b : 1 ≤ a < b ≤ n, we say that [a, b] is a block of σ if {σ(a), . . . , σ(b)} =
{c, . . . , d} for some integer c, d ∈ [n]. A block partition of σ is a partition of [n] into k
blocks [ai, bi] such that (∪i [ai, bi]) ∩ N = [n]. For such a partition, for each i = 1, . . . , k,
consider a permutation σi ∈ Sbi−ai−1 obtained as an order-isomorphic permutation when
restricting σ on [ai, bi]. For each i, let qi ∈ [ai, bi] be a representative element of i. The
permutation ˜σ ∈ [k]k that is order-isomorphic to {σ(q1), . . . , σ(qk)} is called a skeleton of
the block partition. We may view σ as a deflation ˜σ[σ1, . . . , σk].

A permutation σ is d-decomposable if σ = (1), or σ = ˜σ[σ1, . . . , σd] for some d′ ≤ d
and each permutation σi is d-decomposable (we refer to [13] for alternative definitions).
Permutations that are 2-decomposable are called separable [7], and this class includes
preorder traversal sequences [49] as a special case.

To show Theorem 5, it is sufficient to define a reference tree T and a one-finger strategy
for serving a d-decomposable sequence X in T with cost O(log d) · |X|. (Appendix C.)

Combined with the Iacono-Langerman result [28] that Greedy BST has the lazy finger
property, we conclude that the cost of Greedy on any d-decomposable sequence X is at most
O(log d) · |X|. The result is tight and strengthens our earlier bound [13] of |X| · 2O(d2).

Application 3. Generalized monotone sequences. A sequence X ∈ [n]m is k-monotone, if
it can be partitioned into k subsequences (not necessarily contiguous), all increasing or all
decreasing. This property has been studied in the context of adaptive sorting, and special-
purpose structures have been designed to exploit the k-monotonicity of input sequences (see
e.g. [41, 36]). Our results show that BSTs can also adapt to such structure.

Theorem 11. Let X be a k-monotone sequence. Then Fk(X) = O(k · |X|).
It follows that $\text{OPT}(X) \leq O(k \log k) \cdot |X|$ for $k$-monotone sequences. The simulation is straightforward. Let $\{X_1, \ldots, X_k\}$ be a partitioning of $X$ into increasing sequences (such a partition can be found online). Let $T$ be an arbitrary static BST over $[n]$. Consider $k$ fingers $f_1, \ldots, f_k$, initially all on 1. For accessing $x_j \in X_i$, move finger $f_i$ to $x_j$. Observe that over the entire sequence $X$, each finger does only an in-order traversal of $T$, taking $O(n)$ steps. Thus, $F^T_k(X) = O(nk)$.

A lower bound of $\Omega(n \log k)$ follows from enumerative results: for sufficiently large $n$, the number of $k$-monotone permutations $X \in [n]^n$ is at least $k^{\Omega(n)}$ (implied by e.g. [45]). Therefore, by a standard information-theoretic argument (see e.g. [5, Thm. 4.1]), there exists a $k$-monotone permutation $X \in [n]^n$ with $\text{OPT}(X) = \Omega(n \log k)$.

**Further results.** We state our hierarchy result (Theorem 6), also implying a weak separation between $k$-finger bounds and “monotone” bounds.

▷ **Theorem 12** (Appendix E). For all $k$ and infinitely many $n$, there is a $k$-monotone sequence $S_k$ of length $n$, such that:

- $F^{k-1}(S_k) = \Omega(\frac{n}{k} \log(n/k))$
- $F^k(S_k) = O(n)$ (independent of $k$).

In addition, we show a separation between the $k$-finger property and the working set property, showing that for all $k$ and infinitely many $n$, there are sequences $S$ and $S'$ of length $n$, such that $\text{WS}(S) = o(F^k(S))$, and $F^k(S') = o(\text{WS}(S))$. (Appendix F.)

**References**


The result holds, in fact, for the more general case, when each $X_i$ is either increasing or decreasing.


A Offline BST simulation

A.1 BST simulation of a deque

Lemma 13. The minimum and maximum element from a BST-based deque can be deleted in $O(1)$ amortized operations.

Proof. The simulation is inspired by the well-known simulation of a queue by two stacks with constant amortized time per operation ([39, Exercise 3.19]). We split the deque at some position (determined by history) and put the two parts into structures that allow us to access the first and the last element of the deque. It is obvious how to simulate the deque operations as long as the sequences are non-empty. When one of the sequences becomes empty, we split the other sequence at the middle and continue with the two parts. A simple potential function argument shows that the amortized cost of all deque operations is constant. Let $\ell_1$ and $\ell_2$ be the length of the two sequences, and define the potential $\Phi = |\ell_1 - \ell_2|$. As long as neither of the two sequences are empty, for every insert and delete operation both the cost and the change in potential are $O(1)$. If one sequence becomes empty, we split the remaining sequence into two equal parts. The decrease in potential is equal to the length of the sequence before the splitting (the potential is zero after the split). The cost of splitting is thus covered by the decrease of potential.

The simulation by a BST is easy. We realize both sequences by chains attached to the root. The right chain contains the elements in the second stack with the top element as the right child of the root, the next to top element as the left child of the top element, and so on.

A.2 Extended hand

To describe the simulation precisely, we borrow terminology from [18, 19]. Let $T$ be a BST with a set $F$ of $k$ fingers $f_1, \ldots, f_k$. For convenience we assume the root of $T$ to be one of the fingers. Let $S(T, F)$ be the Steiner tree with terminals $F$. A knuckle is a connected component of $T$ after removing $S(T, F)$, i.e. a hanging subtree of $T$. Let $P(T, F)$ be the union of fingers and the degree-3 nodes in $S(T, F)$. We call $P(T, F)$ the set of pseudo fingers. A tendon $\tau_{x,y}$ is the path connecting two pseudo fingers $x, y \in P(T, F)$ (excluding $x$ and $y$) such that there is no other $z \in P(T, F)$ inside. We assume that $x$ is an ancestor of $y$. 
In [18], the hand is defined only over the pseudofingers.

The next terms are new. For each tendon \( \tau_{x,y} \), there are two half tendons, \( \tau_{x,y}^< \) and \( \tau_{x,y}^> \) containing all elements in \( \tau_{x,y} \) which are less than \( y \) and greater than \( y \) respectively. Let \( H(T,F) = \{ \tau_{x,y}^<, \tau_{x,y}^> \mid \tau_{x,y} \text{ is a tendon} \} \) be the set of all half tendons.

For each \( \tau \in H(T,F) \), we can treat \( \tau \) as an interval \([\min(\tau), \max(\tau)]\) where \( \min(\tau), \max(\tau) \) are the minimum and maximum elements in \( \tau \) respectively. For each \( f \in P(T,F) \), we can treat \( f \) as an trivial interval \([f,f]\).

Let \( E(T,F) = P(T,F) \cup H(T,F) \) be the set of intervals defined by all pseudofingers \( P(T,F) \) and half tendons \( H(T,F) \). We call \( E(T,F) \) an extended hand\(^\dagger\). Note that when we treat \( P(T,F) \cup H(T,F) \) as a set of elements, such a set is exactly \( S(T,F) \). So \( E(T,F) \) can be viewed as a partition of \( S(T,F) \) into pseudofingers and half-tendons. Figure 1 illustrates these definitions.

We first state two facts about the extended hand.

\[\textbf{Lemma 14.} \text{ Given any } T \text{ and } F \text{ where } |F| = k, \text{ there are } O(k) \text{ intervals in } E(T,F).\]

**Proof.** Note that \( |P(T,F)| \leq 2k \) because there are \( k \) fingers and there can be at most \( k \) nodes with degree 3 in \( S(T,F) \). Consider the graph where pseudofingers are nodes and tendons are edges. That graph is a tree. So \( |H(T,F)| = O(k) \) as well.

\[\textbf{Lemma 15.} \text{ Given any } T \text{ and } F, \text{ all the intervals in } E(T,F) \text{ are disjoint.}\]

**Proof.** Suppose that there are two intervals \( \tau, x \in E(T,F) \) that intersect each other. One of them, say \( \tau \), must be a half tendon. Because the intervals of pseudofingers are of length zero and they are distinct, they cannot intersect. We write \( \tau = \{ t_1, \ldots, t_k \} \) where \( t_1 < \cdots < t_k \).

Assume w.l.o.g. that \( t_i \) is an ancestor of \( t_{i+1} \) for all \( i < k \), and so \( t_k \) is an ancestor of a pseudofingers \( f \) where \( t_k < f \).

Suppose that \( x \) is a pseudofinger and \( t_j < x < t_{j+1} \) for some \( j \). Since \( t_j \) is the first left ancestor of \( t_{j+1} \), \( x \) cannot be an ancestor of \( t_{j+1} \) in \( T \). So \( x \) is in the left subtree of \( t_{j+1} \). But then \( t_{j+1} \) is a common ancestor of two pseudofingers \( x \) and \( f \), and \( t_{j+1} \) must be a pseudofinger which is a contradiction.

Suppose next that \( x = \{ x_1, \ldots, x_\ell \} \) is a half tendon where \( x_1 < \cdots < x_\ell \). We claim that either \( [x_1, x_i] \subset [t_j, t_{j+1}] \) for some \( j \) or \( [t_1, t_k] \subset [x_j, x_{j+1}] \) for some \( j' \). Suppose not. Then there exist two indices \( j \) and \( j' \) where \( t_j < x_j < t_{j+1} < x_{j'+1} \). Again, \( x_{j'} \) cannot be an ancestor of \( t_{j+1} \) in \( T \), so \( x_{j'} \) is in the left subtree of \( t_{j+1} \). We know either \( x_{j'} \) is the first left

\[\dagger\text{In [18], the hand is defined only over the pseudofingers.}\]
ancestor of \( x_{j'}+1 \) or \( x_{j'}+1 \) is the first right ancestor of \( x_{j'} \). If \( x_{j'} \) is an ancestor of \( x_{j'}+1 \), then \( x_{j'+1} < t_{j+1} \) which is a contradiction. If \( x_{j'}+1 \) is the first right ancestor of \( x_{j'} \), then \( t_{j+1} \) is not the first right ancestor of \( x_{j'} \) and hence \( x_{j'+1} < t_{j+1} \) which is a contradiction again. Now suppose w.l.o.g. \([x_l, x_r] \subset [t_j, t_{j+1}]\). Then there must be another pseudofinger \( f' \) in the left subtree of \( t_{j+1} \), hence \( \tau \) cannot be a half tendon, which is a contradiction. 

\[\Box\]

### A.3 The structure of the simulating BST

In this section, we describe the structure of the BST \( T' \) that we maintain given a \( k \)-finger BST \( T \) and the set of fingers \( F \).

For each half tendon \( \tau \in H(T, F) \), let \( T_\tau' \) be the tree with \( \min(\tau) \) as a root which has \( \max(\tau) \) as a right child. \( \max(\tau) \)'s left child is a subtree containing the remaining elements \( \tau \setminus \{ \min(\tau), \max(\tau) \} \). We implement a BST simulation of a deque on this subtree as defined in Appendix A.1. By Lemma 15, intervals in \( E(T, F) \) are disjoint and hence they are totally ordered. Since \( E(T, F) \) is an ordered set, we can define \( T_{E_0}' \) to be a balanced BST such that its elements correspond to elements in \( E(T, F) \). Let \( T_{E}' \) be the BST obtained from \( T_{E_0}' \) by replacing each node \( a \) in \( T_{E_0}' \) that corresponds to a half tendon \( \tau \in H(T, F) \) by \( T_\tau' \). That is, suppose that the parent, left child, and right child are \( a_{\text{up}}, a_l \), and \( a_r \) respectively. Then the parent in \( T_{E}' \) of the root of \( T_\tau' \) which is \( \min(\tau) \) is \( a_{\text{up}} \). The left child in \( T_{E}' \) of \( \min(\tau) \) is \( a_l \) and the right child in \( T_{E}' \) of \( \max(\tau) \) is \( a_r \).

The BST \( T' \) has \( T_{E}' \) as its top part and each knuckle of \( T \) hangs from \( T_{E}' \) in a determined way.

**Lemma 16.** Each element corresponding to pseudofinger \( f \in P(T, F) \) has depth \( O(\log k) \) in \( T_{E}' \), and hence in \( T' \).

**Proof.** By Lemma 14, \(|E(T, F)| = O(k)|\). So the depth of \( T_{E_0}' \) is \( O(\log k) \). For each node \( a \) corresponding to a pseudofinger \( f \in P(T, F) \), observe that the depth of \( a \) in \( T_{E}' \) is at most twice the depth of \( a \) in \( T_{E_0}' \) by the construction of \( T_{E}' \). 

\[\Box\]

### A.4 The cost for simulating the \( k \)-finger BST

We finally prove the claim on the cost of our BST simulation, which immediately implies Theorem 2. That is, we prove that whenever one of the fingers in a \( k \)-finger BST \( T \) moves to its neighbor or rotates, we can update the maintained BST \( T' \) to have the structure as described in the last section with cost \( O(\log k) \).

We state two observations which follow from the structure of our maintained BST \( T' \) described in A.3. The first observation follows immediately from Lemma 13.

**Lemma 17.** For any half tendon \( \tau \in H(T, F) \), we can insert or delete the minimum or maximum element in \( T_\tau' \) with cost \( O(1) \) amortized.

Next, it is convenient to define a set \( A \), called active set, as a set of pseudofingers, the roots of knuckles whose parents are pseudofingers, and the minimum or maximum of half tendons.

**Lemma 18.** When a finger \( f \) in a \( k \)-finger BST \( T \) moves to its neighbor or rotates with its parent, the extended hand \( E(T, F) = P(T, F) \cup H(T, F) \) is changed as follows.

1. There are at most \( O(1) \) half tendons \( \tau \in H(T, F) \) whose elements are changed. Moreover, for each changed half tendon \( \tau \), either the minimum or maximum is inserted or deleted. The inserted or deleted element was or will be in the active set \( A \).
2. There are at most \( O(1) \) elements added or removed from \( P(T, F) \). Moreover, the added or removed elements were or will be in the active set \( A \).
Lemma 19. Let \( a \in A \) be an element in the active set. We can move \( a \) to the root with cost \( O(\log k) \) amortized. Symmetrically, the cost for updating the root \( r \) to become some element in the active set is \( O(\log k) \) amortized.

Proof. There are two cases. If \( a \) is a pseudofinger or a root of a knuckle whose parent is pseudofinger, we know that the depth of \( a \) was \( O(\log k) \) by Lemma 16. So we can move \( a \) to root with cost \( O(\log k) \). Next, if \( a \) is the minimum or maximum of a half tendon \( \tau \), we know that the depth of the root of the subtree \( T'_{\tau} \) is \( O(\log k) \). Moreover, by Lemma 17, we can delete \( a \) from \( T'_{\tau} \) (make \( a \) a parent of \( T'_{\tau} \)) with cost \( O(1) \) amortized. Then we move \( a \) to root with cost \( O(\log k) \) worst-case. The total cost is then \( O(\log k) \) amortized. The proof for the second statement is symmetric. \( \frown \)

Lemma 20. When a finger \( f \) in a \( k \)-finger BST \( T \) moves to its neighbor or rotates with its parent, the BST \( T' \) can be updated accordingly with cost \( O(\log k) \) amortized.

Proof. According to Lemma 18, we separate our cost analysis into two parts.

For the first part, let \( a \in A \) be the element to be inserted into a half tendon \( \tau \). By Lemma 19, we move \( a \) to root with cost \( O(\log k) \) and then insert \( a \) as a minimum or maximum element in \( T'_{\tau} \) with cost \( O(\log k) \). Deleting \( a \) from some half tendon with cost \( O(\log k) \) is symmetric.

For the second part, let \( a \in A \) be the element to be inserted into a half tendon \( \tau \). By Lemma 19 again, we move \( a \) to root and move back to the appropriate position in \( T'_{E_0} \) with cost \( O(\log k) \). We also need rebalance \( T'_{E_0} \) but this also takes cost \( O(\log k) \). \( \frown \)

Finally, we describe the BST simulation of a \( k \)-finger execution with overhead \( O(\log k) \). Let \( A \) be an arbitrary \( k \)-finger execution in BST \( T \). Whenever there is an update in \( T \) (i.e. a finger moves to its neighbor or rotates), we update the BST \( T' \) according to Lemma 20 with cost \( O(\log k) \) amortized. The BST \( T' \) is maintained so that its structure is as described in Appendix A.3. By Lemma 16, we can access any finger \( f \) of \( T \) from the root of \( T' \) with cost \( O(\log k) \). Therefore, the cost of the BST execution is at most \( O(\log k) \) times the cost of \( A \). This concludes the proof.

B. Missing proofs for Application 1

B.1 Proof of Lemma 9

Part (i).
Suppose that there is some time \( t \) when there are \( \ell' > \ell \) edges \( \{(j_k, x_j), (i_k, s_i)\}_{k=1}^{\ell'} \) such that \( j_k < t \leq i_k \) for all \( k \leq \ell' \). Since each node has a unique parent, \( i_1, \ldots, i_{\ell'-1}, i_{\ell'} \) must be distinct and hence \( \max_{1 \leq k \leq \ell'} j_k \geq \ell + \ell' - 1 \geq t + \ell \). Thus \( \max_{1 \leq k \leq \ell'} j_k \geq t \), a contradiction.

Part (ii).
By construction, the cores are edge-disjoint, and every vertex belongs to some core (the recurrence ends on singleton vertices only). It remains to show that when \( \text{decompose}(H, 0) \) is called during the execution of \( \text{decompose}(T, \ell) \), \( H \) has no edges, i.e. there is no \( i \)-core or \( i \)-body with \( i \leq 0 \).

To see this, define the sequence of graphs \( H_0, \ldots, H_{\ell} \) where \( H_{\ell} = T(\ell, T, X) \), \( H_{i-1} \) is a connected component of \( H_i \setminus C(H_i) \), and \( H_0 = H \). Recall that \( \text{span}(K) \) denotes the time-span of \( K \). By definition of \( C(H_i) \), we have \( \text{span}(H_{i-1}) \subseteq \text{span}(H_i) \).
Suppose for contradiction that $H_0$ has an edge. Denote $\text{span}(H_0) = (t_1, t_2)$, where $t_1 < t_2$. For all $0 \leq i \leq \ell$, it holds that $\text{span}(H_i) \supseteq (t_1, t_2)$. Let $t \in (t_1, t_2)$. We have that $C(H_i)$ contains an edge $((a_i, x_a), (b_i, x_b))$ where $a_i < t < b_i$ for all $0 \leq i \leq \ell$. Since $C(H_i)$ are edge-disjoint, this contradicts part (i).

**Part (iii).**

Suppose there are $i$ active $(i - 1)$-bodies $H'_1, \ldots, H'_i$ of $H$ at time $t$. Since $H$ is an $i$-body, there are $\ell - i$ ancestors $A_1, \ldots, A_{\ell-i}$ of $H$. For each of the cores $C \in \{C(H'_1), \ldots, C(H'_i), C(H), C(A_1), \ldots, C(A_{\ell-i})\}$ which is a set of size $\ell + 1$, there is an edge $(a, s_a), (b, s_b)$ where $a < t \leq b$. This contradicts part (i).

**Part (iv).**

We construct the decomposition greedily. Consider the $(i - 1)$ bodies $H'$ ordered by $\text{start}(H')$ and put $H'$ into the group $H'_j$ for the smallest index $j$ such that the time-span of $H'$ is disjoint from the time-spans of all members of the group. Assume that this process opens up $i' > i - 1$ groups. Then there are $(i - 1)$-bodies $H'_1$ to $H'_{i'}$ (one per group) such that the time-span of the $i$-body $H$ intersects the time-spans of $H'_1$ to $H'_{i'}$, contradicting part (iii).

**B.2 Proof of Lemma 10**

We analyze the total cost of calling $\text{access}(H, F, u)$ for all $u \in V(H)$. The total cost due to recursive calls in Step 2 is accounted by the term $\sum_{H' \in \mathcal{H}} \text{cost}(H')$. The remaining operations amount to moving $n_f(i)$ fingers from $\text{start}(H)$ to $\text{end}(H)$ and back, along the $i$-core $C(H)$. The cost of this is exactly $2 \cdot n_f(i) \cdot w_T(C(H))$. In addition we need to traverse, using $n_f(i - 1)$ fingers, the edges connecting $C(H)$ to $\text{start}(H')$, twice for all $H' \in \mathcal{H}$. The total cost thus becomes at most $2 \cdot n_f(i) \cdot w_T(C^+(H)) + \sum_{H' \in \mathcal{H}} \text{cost}(H')$.

We argue now by induction that for an $i$-body $H$, we have $\text{cost}(H) \leq 2 \cdot i! \cdot w_T(H)$. For $i = 1$, $H = C(H) = C^+(H)$. Thus, by the inductive step:

$$\text{cost}(H) \leq 2 \cdot n_f(1) \cdot w_T(C^+(H)) \leq 2 \cdot w_T(H).$$

For the general inductive step:

$$\text{cost}(H) \leq 2 \cdot n_f(i) \cdot w_T(C^+(H)) + \sum_{H' \in \mathcal{H}} \text{cost}(H')$$

$$\leq 2 \cdot i! \cdot w_T(C^+(H)) + \sum_{H' \in \mathcal{H}} 2 \cdot (i - 1)! \cdot w_T(H')$$

$$\leq 2 \cdot i! \cdot \left( w_T(C^+(H)) + \sum_{H' \in \mathcal{H}} w_T(H') \right)$$

$$= 2 \cdot i! \cdot w_T(H).$$

**C Decomposable Sequences**

**Lemma 21.** Let $X = (x_1, \ldots, x_n)$ be a $k$-decomposable permutation of length $n$. Then $F^4_k(X) \leq 4(|X| - 1) \left\lceil \log k \right\rceil$.

**Proof.** It is sufficient to define a reference tree $T$ for which $F^4_k(X)$ achieves such bound. We remark that the tree will have auxiliary elements. We construct $T$ recursively. If $X$ has length one, then $T$ has a single node and this node is labeled by the key in $X$. Clearly, $F^4_k(X) = 0$. 
Otherwise, let $X = \tilde{X}[X_1, \ldots, X_j]$ with $j \in [k]$ be the outermost partition of $X$. Denote by $T_i$ the tree for $X_i$ that has been inductively constructed. Let $T_0$ be a BST of depth at most $\lceil \log j \rceil$ and with $j$ leaves. Identify the $i$-th leaf with the root of $T_i$ and assign keys to the internal nodes of $T_0$ such that the resulting tree is a valid BST. Let $r_i$ be the root of $T_i$, $0 \leq i \leq j$ and let $r = r_0$ be the root of $T$. Then

$$d_T(r, x_1) \leq \lceil \log k \rceil + d_{T_1}(r_1, x_1)$$

$$d_T(r, x_n) \leq \lceil \log k \rceil + d_{T_j}(r_j, x_n)$$

$$d_T(x_{i-1}, x_i) \leq \begin{cases} 
\lceil \log k \rceil + d_{T_k}(x_{i-1}, x_i) & \text{if } x_{i-1}, x_i \in X_t \\
2\lceil \log k \rceil + d_{T_i}(r_i, x_{i-1}) + d_{T_{i+1}}(r_{i+1}, x_i) & \text{if } x_{i-1} \in X_t \text{ and } x_i \in X_{i+1}, 
\end{cases}$$

and hence

$$F_T^2(X) = d_T(r, x_0) + \sum_{1 \geq 2} d_T(x_{i-1}, x_i) + d_T(x_n, r)$$

$$\leq 2j \lceil \log k \rceil + \sum_{1 \leq i \leq j} F_T^2(X_i) \leq 2j \lceil \log k \rceil + \sum_{1 \leq i \leq j} 4(|X_i| - 1) \lceil \log k \rceil$$

$$\leq (2j - 4j + 4 \sum_{1 \leq i \leq j} |X_i|) \lceil \log k \rceil \leq 4(|X| - 1) \lceil \log k \rceil,$$

where the last inequality uses $j \geq 2$. ▷

### D Finger bounds with auxiliary elements

Recall that $F(X)$ is defined as the minimum over all BSTs $T$ on $[n]$ of $F_T(X)$. It is convenient to define a slightly stronger finger bound that also allows auxiliary elements. Define $\hat{F}(X)$ as the minimum over all BSTs $T$ that contain the keys $[n]$ (but the size of $T$ can be much larger than $n$). We define $\hat{F}_k(X)$ as the $k$-finger bound when the tree is allowed to have auxiliary elements. We argue that the two definitions are equivalent.

► Theorem 22. For any integer $k$, $F_k(X) = \Theta(\hat{F}_k(X))$ for all $X$.

Proof. It is clear that $\hat{F}_k(X) \leq F_k(X)$. We only need to show the converse.

Let $T$ be a BST (with auxiliary elements) such that $F_T(X) = \hat{F}_k(X)$. Denote by $\vec{f}$ the optimal finger strategy on $T$. Let $[n] \cup Q$ be the elements of $T$ where $Q$ is the set of auxiliary elements in $T$. For each $a \in [n] \cup Q$, let $d_T(a)$ be the depth of key $a$ in $T$, and let $w(i) = 4^{-d_T(i)}$. For any two elements $i$ and $j$ and set $Y \subseteq [n] \cup Q$, let $w_Y[i : j]$ be the sum $\sum_{k \in Y \cap [i,j]} w(k)$ of the weights of the elements in $Y$ between $i$ and $j$ (inclusive). For any $i, j \in [n] \cup Q$ such that $i \leq j$, we have

$$\log \frac{w_{[n] \cup Q}[i : j]}{\min(w(i), w(j))} = O(d_T(i, j)),$$

where $d_T(i, j)$ is the distance from $i$ to $j$ in $T$. So, this same bound also holds when considering only keys in $[n]$. That is, for $i, j \in [n]$, we have

$$\log \frac{w_{[n]}[i : j]}{\min(w(i), w(j))} = O(d_T(i, j)).$$
Given the weight of \( \{w(a)\}_{a \in [n]} \), the BST \( T^* \) (without auxiliary elements) is constructed by invoking Lemma 23. We bound the term \( F^k_T(X) \) (using strategy \( \hat{f} \)) by

\[
O\left( \sum_{t} d_T(x_{\sigma(f,t)}, x_t) \right) = O\left( \sum_{t=1}^{m-1} \frac{w[n][x_t : x_{\sigma(f,t)}]}{\min(w(x_t), w(x_{\sigma(f,t)}))} \right)
\]

\[
= O\left( \sum_{t=1}^{m-1} d_T(x_{\sigma(f,t)}, x_t) \right) = O(F^k_T(X))
\]

where \( X = (x_1, \ldots, x_m) \). Therefore, \( F^k(X) \leq F^k_T(X) = O(F^k_T(X)) = O(F^k(X)) \).

\[\blacktriangleleft\]

**Lemma 23.** Given a weight function \( w(\cdot) \), and \( W = \sum_{i \in [n]} w(i) \), there is a deterministic construction of a BST \( T_w \) such that the depth of every key \( i \in [n] \) is \( d_{T_w}(i) = O(\log \frac{W}{w(i)}) \).

**Proof.** Let \( w_1, \ldots, w_n \) be a sequence of weights. We show how to construct a tree in which the depth of element \( \ell \) is \( O(\log w[1 : \ell]/\min(w_1, w_\ell)) \).

For \( i \geq 1 \), let \( j_i \) be minimal such that \( w[1 : j_i] \geq 2^i w_1 \). Then \( w[1 : j_i - 1] < 2^i w_1 \) and \( w[j_i - 1 + 1 : j_i] \leq 2^{i-1} w_1 + w_{j_i} \).

Let \( T_i \) be the following tree. The right child of the root is the element \( j_i \). The left subtree is a tree in which element \( \ell \) has depth \( O(\log 2^{i-1} w_1/w_{j_i}) \).

The entire tree has \( w_1 \) in the root and then a long right spine. The trees \( T_i \) hang off the spine to the left. In this way the depth of the root of \( T_i \) is \( O(i) \).

Consider now an element \( \ell \) in \( T_i \). Assume first that \( \ell \neq j_i \). The depth is

\[
O \left( i + \log \frac{2^{i-1} w_1}{w_\ell} \right) = O \left( i + \log \frac{2^{i-1} w_1}{\min(w_1, w_\ell)} \right)
\]

\[
= O \left( \log \frac{2^{i-1} w_1}{\min(w_1, w_\ell)} \right) = O \left( \frac{w[1 : \ell]}{\min(w_1, w_\ell)} \right).
\]

For \( \ell = j_i \), the depth is

\[
O(i) = O \left( \log \frac{2^i w_1}{w_1} \right) = O \left( \log \frac{w[1 : j_i]}{\min(w_1, w_{j_i})} \right).
\]

\[\blacktriangleleft\]

**E Proof of Theorem 6**

Let \( n \) be an integer multiple of \( k \) and \( \ell = n/k \). Consider the tilted \( k \)-by-\( \ell \) grid \( S_k \). More precisely, the access sequence is defined as: \( 1, \ell + 1, \ldots, \ell \cdot (k - 1) + 1, 2, \ell + 2, \ldots, (k - 1)\ell + 2, \ldots, (k - 1)\ell + \ell \). We denote the elements of \( S_k \) as \( s_i \), for \( i = 1, \ldots, n \). To see the geometry of this sequence, one may view it as a partitioning of the keys \([n]\) into “blocks” \( B_i : i = 1, \ldots, k \) where \( B_i \) contains the keys in \( \{\ell(i-1) + 1, \ell(i-1) + 2, \ldots, \ell i\} \), so we have \( |B_i| = \ell \) and \( \bigcup_{i=1}^k B_i = [n] \). The sequence \( S_k \) consists of an interleaving of an increasing traversal of each block.

**Lemma 24.** \( F^k(S_k) = O(n) \).

**Proof.** The main idea is to use each finger to serve only the keys inside blocks and to use a separate finger for each block. (recall that there are \( k \) blocks and \( k \) fingers.) We create a reference tree \( T \) and argue that \( F^k_T(S_k) = O(n) \). Let \( T_0 \) be a BST of height \( O(\log k) \) and with \( k \) leaves. Each leaf of \( T_0 \) corresponds to the keys \( \{\ell \cdot i + \frac{1}{2}\}_{i=1}^k \). The non-leaves of \( T_0 \) are assigned arbitrary fractional keys that are consistent with the BST properties. For
each $i \in [k]$, path $P_i$ is defined as a BST with key $\ell \cdot (i - 1) + 1$ (i.e. the smallest key in block $B_i$) at the root, where for each $j = 0, \ldots, (\ell - 1)$, the key $\ell(i - 1) + j$ has $\ell(i - 1) + (j + 1)$ as its only (right) child. The final tree $T$ is obtained by hanging each path $P_i$ as a left subtree of a leaf $\ell \cdot (i - 1) + \frac{1}{2}$. The $k$-finger strategy is simple: The $i$-th finger only takes care of the elements in block $B_i$. The cost for the first access in block $B_i$ is $O(\log k)$, and afterwards, the cost is only $O(1)$ per access. So the total access cost is $O(\frac{n}{k} \log k + n) = O(n)$.

The rest of this section is devoted to proving the following:

\textbf{Theorem 25.} $F^{k-1}(S_k) = \Omega(\frac{n}{k} \log(n/k))$

Let $T$ be an arbitrary reference tree. We argue that $F^{k-1}(S_k) = \Omega(\frac{n}{k} \log(n/k))$.

A finger configuration $f = (f(1), \ldots, f(k-1)) \in [n]^{k-1}$ specifies to which keys the fingers are currently pointing. Any finger strategy can be described by a sequence $f_1, \ldots, f_n$, where $f_t$ is the configuration after element $s_t$ is accessed. As before, we assume w.l.o.g. the following lazy update strategy:

\textbf{Lemma 26.} For each time $t$, the configurations $f_t$ and $f_{t+1}$ differ at exactly one position. In other words, we only move the finger that is used to access $s_{t+1}$.

We view the input sequence $S_k$ as having $\ell$ phases: The first phase contains the subsequence $1, \ell + 1, \ldots, (\ell - 1) + 1$, and so on. Each phase is a subsequence of length $k$ that accesses keys starting in block $bset_1$ and so on, until the block $B_k$.

\textbf{Lemma 27.} For each phase $t \in \{1, \ldots, \ell\}$, there is a time $t \in [(p-1)k+1, p\cdot k]$ such that $s_t$ is accessed by finger $j$ such that $f_{t-1}(j)$ and $f_t(j)$ are in different blocks, and $f_{t-1}(j) < f_t(j)$. That is, this finger moves to the block $B_b$, $b = t \mod k$, from some block $B_{b'}$, where $b' < b$, in order to serve $s_t$.

\textbf{Proof.} Consider the accesses in blocks $B_1, \ldots, B_k$ in order. After the access in $B_1$, we have a finger in $B_1$ and hence at most $k - 2$ fingers in blocks $B_2, \ldots, B_k$. If the access to $B_2$ is served by a finger being in block $B_1$ before the access, we are done. Otherwise, it is served by a finger being in blocks $B_{2,2}$ before the access. Then we have two fingers in blocks $B_{2,2}$ after the access and at most $k - 3$ fingers in blocks $B_{2,3}$. Continuing in this way, we will find the desired access.

For each phase $t \in [\ell]$, let $t_p$ denote the time for which such a finger moves across the blocks from left to right; if they move more than once, we choose $t_p$ arbitrarily. Let $J = \{t_p\}_{p=1}^{\ell}$. For each finger $j \in [k-1]$, each block $i \in [k]$ and block $i' \in [k]: i < i'$, let $J(j, i, i')$ be the set containing the time $t$ for which finger $f(j)$ is moved from block $B_i$ to block $B_{i'}$ to access $s_t$. Let $c(j, i, i') = |J(j, i, i')|$. Notice that $\sum_{j,i,i'} c(j,i,i') = \frac{n}{k} = \ell$, due to the lemma. Let $P(j, i, i')$ denote the phases $p$ for which $t_p \in J(j, i, i')$.

\textbf{Lemma 28.} $\sum_{j,i,i': c(j,i,i') \geq 16} c(j,i,i') \geq n/2k$ if $n = O(k^2)$.

\textbf{Proof.} There are only at most $k^3$ triples $(j, i, i')$, so the terms for which $c(j, i, i') < 16$ contribute to the sum at most $16k^3$. This means that the sum of the remaining is at least $n/k - 16k^3 \geq n/2k$ if $n$ satisfies $n = O(k^2)$.

From now on, we consider the sets $J'$ and $J'(j, i, i')$ that only concern those $c(j, i, i')$ with $c(j, i, i') \geq 16$ instead.

\textbf{Lemma 29.} There is a constant $\eta > 0$ such that the total access cost during the phases $P(j,i,i')$ is at least $\eta c(j,i,i') \log c(j,i,i')$.\hfill$\blacksquare$
Once we have this lemma, everything is done. Since the function \( g(x) = x \log x \) is convex, we apply Jensen’s inequality to obtain:

\[
\frac{1}{|J'|} \sum_{j,i,i'} \eta(c(j,i,i')) \geq \eta \cdot \frac{n}{2k|J'|} \log(n/2k|J'|).
\]

Note that the left side is the term \( E[g(x)] \), while the right side is \( g(E(x)) \). Therefore, the total access cost is at least \( \frac{mn}{\eta} \log(n/2k) \). We now prove the lemma.

**Proof of Lemma 29.** We recall that, in the phases \( P(j,i,i') \), the finger-\( j \) moves from block \( B_i \) to \( B_{i'} \) to serve the request at corresponding time. For simplicity of notation, we use \( \bar{J} \) and \( C \) to denote \( J(j,i,i') \) and \( c(j,i,i') \) respectively. Also, we use \( \bar{f} \) to denote the finger-\( j \).

For each \( t \in \bar{J} \), let \( a_t \in B_i \) be the key for which the finger \( \bar{f} \) moves from \( a_t \) to \( s_{a_t} \) when accessing \( s_{a_t} \in B_{i'} \). Let \( \bar{J} = \{ t_1, \ldots, t_{C} \} \) such that \( a_{t_1} < a_{t_2} < \ldots < a_{t_C} \). Let \( R \) be the lowest common ancestor in \( T \) of keys in \( [a_{t_{C/2}}^{+1}, a_{t_C}] \).

**Lemma 30.** For each \( r \in \{1, \ldots, [C/2]\} \), the access cost of \( s_{a_r} \) and \( s_{a_{C-r}} \) is together at least \( \min\{d_T(R, s_{a_r}), d_T(R, s_{a_{C-r}})\} \).

**Proof.** Let \( u_r \) be the lowest common ancestor between \( a_{t_r} \) and \( s_{a_{t_r}} \). Then the cost of accessing \( s_{a_r} \) is at least \( d_T(u_r, s_{a_r}) \). If \( s_{a_r} \) is in the subtree rooted at \( R \), then \( u_r \) must be an ancestor of \( R \) (because \( a_{t_r} < a_{t_{C/2}} < a_{t_C} < s_{a_{t_r}} \)) and hence \( d_T(u_r, s_{a_r}) \geq d_T(R, s_{a_r}) \). Thus the cost it at least \( d_T(R, s_{a_r}) \). Otherwise, we know that \( s_{a_r} \) is outside of the subtree rooted at \( R \), and so is \( s_{a_{C-r}} \). On the other hand, \( a_{t_{C-r}} \) is in such subtree, so moving the finger from \( a_{t_{C-r}} \) to \( s_{a_{C-r}} \) must touch \( R \), therefore costing at least \( d_T(R, s_{a_{C-r}}) \).

Lemma 30 implies that, for each \( r = 1, \ldots, [C/2] \), we pay the distance between some element \( v_r \in \{s_{a_r}, s_{a_{C-r}}\} \) to \( R \). The total such costs would be \( \sum_r d_T(R, v_r) \). Applying the fact that (i) \( v_r \)’s are different and (ii) there are at most \( 3^d \) vertices at distance \( d \) from a vertex \( R \), we conclude that this sum is at least \( \sum_r d_T(R, v_r) \geq \Omega(C \log C) \).

**F Working set and \( k \)-finger bounds are incomparable**

We show the following theorem.

**Theorem 31.**

1. There exists a sequence \( S \) such that \( WS(S) = o(F_k(S)) \), and
2. There exists a sequence \( S' \) such that \( F_k(S') = o(WS(S')) \).

The sequence \( S' \) above is straightforward: For \( k = 1 \), just consider the sequential access \( 1, \ldots, n \) repeated \( mn \) times. For \( m \) large enough, the working set bound is \( \Omega(m \log n) \). However, if we start with the finger on the root of the tree which is just a path, then the lazy finger bound is \( O(m) \). The \( k \)-finger bound is always less than lazy finger bound, so this sequence works for the second part of the theorem.

The existence of the sequence \( S \) is slightly more involved (the special case for \( k = 1 \) was proved in [8]), and is guaranteed by the following theorem, the proof of which comprises the remainder of this section.

**Theorem 32.** For all \( k = O(n^{1/2-\epsilon}) \), there exists a sequence \( S \) of length \( m \) such that \( WS(S) = O(m \log k) \) whereas \( F_k(S) = \Omega(m \log(n/k)) \).
We construct a random sequence $S$ and show that while $\text{WS}(S) = O(m \log k)$ with probability one, the probability that there exists a tree $T$ such that $P^k_T(S) \leq cm \log_3(n/k)$ is less than $1/2$ for some constant $c < 1$. This implies the existence of a sequence $S$ such that for all trees $T$, $P^k_T(S) = \Omega(m \log(n/k))$.

The sequence is as follows. We have $Y$ phases. In each phase we select $2k$ elements $R_i = \{r_j^{1:2k}\}_{j=1}^i$ uniformly at random from $[n]$. We order them arbitrarily in a sequence $S_i$, and access $|S_i|^{X/2k}$ (access $S_i$, $X/2k$ times). The final sequence $S$ is a concatenation of the sequences $|S_i|^{X/2k}$ for $1 \leq i \leq Y$. Each phase has $X$ accesses, for a total of $m = XY$ accesses overall. We will choose $X$ and $Y$ appropriately later.

**Working set bound.** One easily observes that $\text{WS}(S) = O(Y(2k \log n + (X - 2k) \log(2k)))$, because after the first $2k$ accesses in a phase, the working set is always of size $2k$. We choose $X$ such that the second term dominates the first, say $X \geq 5k \log_{\log 2k} n$. We then have that the working set bound is $O(XY \log k) = O(m \log k)$, with probability one.

**k-finger bound.** Fix a BST $T$. We classify the selection of the set $R_i$ as being $d$-good for $T$ if there exists a pair $r_j^{1}, r_j^{2} \in R_i$ such that their distance in $T$ is less than $d$. The following lemma bounds the probability of a random selection being $d$-good for $T$.

**Lemma 33.** Let $T$ be any BST. The probability that $R_i$ is $d$-good for $T$ is at most $8k^23^d/n$.

**Proof.** We may assume $8k^23^d/n < 1$ as the claim is void otherwise. We compute the probability that a selection $R_i$ is not $d$-good first. This happens if and only if the balls of radius $d$ around every element $r_j^{j}$ are disjoint. The volume of such a ball is at most $3^d$, so we can bound this probability as

$$P[R_i \text{ is not } d\text{-good for } T] = \prod_{i=1}^{2k} \left(1 - \frac{i3^d}{n}\right) \geq \left(1 - \frac{2i3^d}{n}\right)^{2k} = 1 - \exp \left(2k \ln \left(1 - \frac{2k3^d}{n}\right)\right) \leq 1 - \exp \left(-8k^23^d/n\right) \leq 8k^23^d/n,$$

where the last two inequalities follow from $\ln(1 - x) > -2x$ for $x \leq 1/2$ (note that $8k^23^d/n < 1$ implies $2k3^d/n \leq 1/2$) and $e^x > 1 + x$, respectively. ▷

Observe that if $R_i$ is not $d$-good, then the $k$-finger bound of the access sequence $|S_i|^{X/2k}$ is $\Omega(dX - k)) = \Omega(dX)$. This is because in every occurrence of $S_i$, there will be some $k$ elements out of the $2k$ total that will be outside the $d$-radius balls centered at the current $k$ fingers.

We call the entire sequence $S$ $d$-good for $T$ if at least half of the sets $R_i$ are $d$-good for $T$. Thus if $S$ is not $d$-good, then $P^k_T(S) = \Omega(XYd)$.

**Lemma 34.** $P[S \text{ is } d\text{-good for } T] \leq \left(\frac{8k^23^d}{n}\right)^{Y/2}$. ▶
Proof. By the previous lemma and by definition of goodness of $S$, we have that
\[ P[S \text{ is } d\text{-good for } T] \leq \binom{Y}{Y/2} \left( \frac{8k^23^d}{n} \right)^{Y/2} \leq 4^{Y/2} \left( \frac{8k^23^d}{n} \right)^{Y/2} = \left( \frac{32k^23^d}{n} \right)^{Y/2}. \]

The theorem now follows easily. Taking a union bound over all BSTs on $[n]$, we have
\[ P[S \text{ is } d\text{-good for some BST } T] \leq 4^n \left( \frac{32k^23^d}{n} \right)^{Y/2}. \]

Now set $Y = 2n$. We have that
\[ P[\exists \text{ a BST } T: F^k_T(S) \leq md/4] \leq 4^n \left( \frac{32k^23^d}{n} \right)^n. \]

Putting $d = \log_3 \frac{n}{256k^2}$ gives that for some constant $c < 1$,
\[ P[\exists \text{ a BST } T: F^k_T(S) \leq c(m \log(n/k))] \leq 4^n \left( \frac{32k^23^d}{n} \right)^n = 1/2 \]

which implies that with probability at least 1/2 one of the sequences in our random construction will have $k$-finger bound that is $\Omega(m \log(n/k))$. The working set bound is always $O(m \log k)$. This establishes the theorem.
On Counting Oracles for Path Problems

Ivona Bezáková
Department of Computer Science, Rochester Institute of Technology, Rochester, NY, USA
ib@cs.rit.edu

Andrew Searns
Rochester Institute of Technology, Rochester, NY, USA
abs2157@rit.edu

Abstract
We initiate the study of counting oracles for various path problems in graphs. Distance oracles have gained a lot of attention in recent years, with studies of the underlying space and time tradeoffs. For a given graph $G$, a distance oracle is a data structure which can be used to answer distance queries for pairs of vertices $s, t \in V(G)$. In this work, we extend the setup to answering counting queries: for a pair of vertices $s, t$, the oracle needs to provide the number of (shortest or all) paths from $s$ to $t$. We present $O(n^{1.5})$ preprocessing time, $O(n^{1.5})$ space, and $O(\sqrt{n})$ query time algorithms for oracles counting shortest paths in planar graphs and for counting all paths in planar directed acyclic graphs. We extend our results to other graphs which admit small balanced separators and present applications where our oracle improves the currently best known running times.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases Counting oracle, Path problems, Shortest paths, Separators

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.56

Funding Research supported by NSF grant CCF-1319987 and by an REU (Research Experience for Undergraduates) supplement.

1 Introduction

Shortest path problems have been heavily studied for decades and the developed algorithms are among the most important algorithmic building blocks. In the most traditional setup, one is given a graph $G$ and two vertices $s, t$ and the goal is to find a shortest path from $s$ to $t$ in $G$. Due to many applications querying for multiple $s, t$ pairs, the design of so-called distance oracles has gained a lot of attention in recent years [13, 9, 18, 6, 11, 8, 10, 3]. In an oracle approach, for a given graph $G$, the goal is to pre-compute a not too large data structure (an oracle) which can then be used to answer distance queries for pairs of vertices $s, t$ in as fast time as possible. Many previous works, which we discuss in more detail later, have studied the tradeoffs between the required space and the query time for such distance oracles for various graph classes. Among the prime applications of these oracle results is map querying, where a user often prefers knowing not just one of the optimal routes, but they would like to be shown a variety of options. Hence, we propose to amend distance oracles with counting: in addition to the distance from $s$ to $t$, a counting path oracle returns also the number of all shortest paths from $s$ to $t$. Such an oracle can then be used to generate the paths or provide a random sample when the total number of paths is prohibitively large.

© Ivona Bezáková and Andrew Searns; licensed under Creative Commons License CC-BY
Editors: Wen-Lian Hsu, Der-Tsai Lee, and Chung-Shou Liao; Article No. 56; pp. 56:1–56:12
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
We design counting oracles for the following two problems: \texttt{#SHORTPATH-ORACLE}, where one is given a positively weighted graph and the goal is to construct an oracle which answers queries of “how many shortest paths from \(s\) to \(t\) are there?”, and \texttt{#PATH-DAG-ORACLE}, where one is given a directed acyclic graph (DAG) and the oracle answers queries of “how many paths from \(s\) to \(t\) are there?”. We note that the second problem is \#P-hard for general graphs \cite{20}, but both problems can be solved in polynomial time within the specified graph class. The second problem, which has applications of its own described below, helps us build an oracle for the first problem. For both problems, when the input is a planar graph with \(n\) vertices, we design oracles which take \(O(n^{1.5})\) time to construct, take \(O(n^{1.5})\) space, and each query can be answered within \(O(\sqrt{n})\) time\(^1\).

A straightforward approach to both problems yields an oracle which takes \(O(n^2)\) space and \(O(1)\) query time, by simply pre-computing all of the possible queries. In a DAG one can compute the number of paths from one vertex to all other vertices in linear time, leading to an \(O(n(n + m))\) preprocessing time to compute the oracle for a graph with \(n\) vertices and \(m\) edges. For planar graphs, this preprocessing time is \(O(n^2)\) since \(m = O(n)\). For several applications, the number of queries can be linear, leading to an \(O(n^2)\) preprocessing time and an overall \(O(n)\) time across all queries in the planar setting. With our results, we speed up the running time for such applications to \(O(n^{1.5} + n\sqrt{n}) = O(n^{1.5})\). For arbitrary positively weighted graphs, one can compute the number of shortest paths from one vertex to all other vertices in polynomial time, typically within the same running time as finding the distances to all other vertices. For example, one can extend the Dijkstra’s algorithm to compute, in addition to the distances, also the respective path counts, and keep updating them throughout the computation. Our results speed up these traditional approaches.

Our techniques employ balanced separators, which are a staple of planar graph algorithms but to the best of our knowledge have not been used for any counting problems. The distance oracle results are ingenious and faster than our results but as far as we see they do not extend to counting. In an optimization problem, one can focus on a certain canonical type of the wanted object, such as a left-most shortest path, or one can even assume that there is a unique shortest path between any pair of vertices. (This can be obtained by small random perturbations of the edge weights.) For a counting problem there appears to be the need for more stored information or longer query time. In particular, we store, for each vertex in the separator, certain path counts to all other vertices in the graph, proceeding in a divide-and-conquer manner on the two parts of the graph. The main technical aspect of our contribution lies in a case analysis that proves that each path has been accounted for exactly once. We generalize our planar results to general graphs which admit small balanced separators.

### 1.1 Related work and applications

Distance oracles have been studied for several decades, with several very recent exciting results. The current state of the art exact distance oracle of Gawrychowski, Mozes, Weimann, and Wulff-Nilsen \cite{13} requires \(O(n^{1.5})\) space and can answer queries in \(O(\log n)\) time. This work improved on a recent result of Cohen-Addad, Dahlgaard, and Wulff-Nilsen \cite{9} who

\(^1\) We note that the returned counts may be exponentially large, for example when the graph is a path where every edge has been duplicated – if \(s\) and \(t\) are the end-points, the number of shortest \(s\)-\(t\) paths is \(2^{n-1}\). Therefore, manipulating the counts can incur an additional \(O(n \text{ polylog } n)\) factor in the running time. To simplify our presentation throughout this paper, we will (slightly optimistically) assume that each arithmetic operation (addition, multiplication of the counts) takes \(O(1)\) time.
designed an oracle with $O(n^{5/3})$ space and $O(\log n)$ query time. Furthermore, both works obtained space/time tradeoffs: for a given $S$, they design an oracle which takes $S$ space and the query time is a function of $S$. In particular, [13] obtain a query time $\tilde{O}(\max\{1, n^{1.5}/S\})$ for $S \in [n, n^2]$, while [9] answer queries within time $\tilde{O}(n^{5/2}S^{3/2})$ for $S \geq n^{3/2}$ (where the $\tilde{O}$ notation hides logarithmic factors). Other previous works, on which the two mentioned results build, also studied distance oracles and their space/time tradeoffs [18, 6, 11, 8, 10, 3]. As far as we see, these results do not extend to counting without significant increase in the running time (or space). Of note is also extensive study of approximate distance oracles, with either relative or absolute error, which can achieve a near-linear space and near-constant query time, see [1] and the references within. As for the counting variant in an approximate distance setting, Mihaláč, Sránek, and Widmayer [17] showed that counting all $s$-$t$ paths up to a given length in a DAG is #P-complete. They also give a fully polynomial-time approximation scheme (FPTAS) for the problem, yielding an approximate counting approximate distance oracle in DAGs. However, the techniques heavily rely on the graph being acyclic. We also note two other hardness results for counting: Yamamoto [21] proved that there is no fully polynomial approximation scheme (FPRAS) for approximately counting all paths in a graph, unless $RP = NP$. On the fixed-parameter tractable side, Flum and Grohe [12] showed that the problem of counting paths of length $k$ is #W[1]-complete.

Among applications of counting oracles for all paths in a DAG is the problem of counting minimum $(s, t)$-cuts in planar and bounded genus graphs. The problem of counting minimum $(s, t)$-cuts has been studied since the 1980’s due to its connection to the $(s, t)$ network reliability problem. Provan and Ball [19] proved that it is #P-complete for general graphs and in [4] they gave a general outline that reduces the problem in planar graphs with both $s$ and $t$ on the outerface to the problem of counting all paths in a planar DAG. Their technique was subsequently generalized to any location of $s$ and $t$ by Bezáková and Friedlander [5] and Chambers, Fox, and Nayyeri [7] further extended the approach to bounded genus graphs. In all these scenarios, one needs to count all paths between $d$ pairs of vertices in a planar or bounded genus DAG. For planar graphs, this results in a running time of $O(n \log n + d n) = O(n^2)$ since $d = O(n)$, which our result improves to $O(n \log n + n^{1.5}) = O(n^{1.5})$ since instead of counting the paths for each pair in $O(n)$ time, we can make queries in $O(\sqrt{n})$ time per pair. The running time encompasses also the oracle preprocessing time. It is worth noting that our attempts to use more advanced decomposition techniques such as the $r$-divisions led to these same running times, making us wonder if a faster than $O(n^{1.5})$ algorithm exists.

The paper is organized as follows. In Section 2 we discuss preliminaries, including how to extend existing single source shortest path (SSSP) algorithms to counting, incurring an additional linear term in the running time. Section 3 presents counting oracles for all paths in planar DAGs, then we discuss counting oracles for shortest paths in positively weighted directed or undirected planar graphs in Section 4, and generalize the oracles beyond planar graphs in Section 5.

### 2 Preliminaries

An undirected graph $G = (V, E)$ is a set of vertices $V$ and edges $E \subseteq (V \times V)$ of unordered pairs. In a directed graph $G = (V, E)$, the edges are ordered pairs and we refer to them as arcs, using the standard convention that $(u, v)$ indicates an arc from vertex $u$ to vertex $v$. Unless specifically noted, our results apply to both directed and undirected graphs. (We phrase all our results for graphs but they can be naturally extended to multigraphs which can
have multiple edges between the same pair of vertices.) A positively weighted graph, denoted by $G = (V, E, w)$, assigns a positive weight $w(e)$ to each edge $e \in E$ (i.e., $w : E \rightarrow \mathbb{R}^+$). For an $S \subseteq V$, we use $G[S]$ to denote the subgraph of $G$ induced by $S$. Throughout this text we use $n = |V|$ to denote the number of vertices and $m = |E|$ the number of edges.

A path $p$ in a graph $G$ is a sequence of vertices $v_1, v_2, \ldots, v_k$, $k \geq 1$, where $(v_i, v_{i+1}) \in E$ for each $i \in \{1, \ldots, k-1\}$. A path with no repeated vertices is called a simple path. For convenience, we refer to a path starting at vertex $v_1$ and ending at vertex $v_k$ as a $v_1$-$v_k$ path.

We define the relation “is before on $G$” by $u \prec_p v_2$ where $u \neq v_1$, $v_2 = v_j$, and $i \leq j$. We say that $v_i, v_{i+1}, \ldots, v_j$, where $i < j$, is a sub-path of a path $p = v_1, \ldots, v_k$. The length of a path $p = v_1, \ldots, v_k$ is $k - 1$ in unweighted graphs, and $\sum_{i=1}^{k-1} w(v_i, v_{i+1})$ in weighted graphs. A $u_1$-$u_2$ path is shortest if its length is the smallest possible across all $u_1$-$u_2$ paths. The length of a shortest $u_1$-$u_2$ path is called the distance from $u_1$ to $u_2$. A cycle $v_1, \ldots, v_k$ is a path where $v_1 = v_k$ and $k \geq 2$. A graph is acyclic if it does not contain any cycles.

\textbf{Observation 1.} Any path in an acyclic graph is simple. Any shortest path in a positively weighted graph is simple. If $p$ is a shortest path in a graph $G$, then any sub-path of $p$ must also be shortest.

We say that a class of graphs admits an $(\alpha, f(n))$-balanced separator, where $f(n)$ is a function and $\alpha$ is a constant, if for every graph $G$ with $n$ vertices its vertices can be partitioned into three sets $A, B, C$ such that the size of $A$ and $B$ are each upper-bounded by $\alpha n$, the size of $C$ is $O(f(n))$, and there are no edges connecting a vertex in $A$ with a vertex in $B$. We will refer to such a separator as an $(A, B, C)$ separator.

A graph is planar if it has a planar embedding, that is, if it can be drawn in a plane without any of its edges crossing one another (except for their end-points). A graph is said to be of genus $g$ if it has a crossing-free embedding into a surface of genus $g$. Planar and bounded genus graphs are sparse, in particular $m = O(n)$ and $m = O(n + g)$, respectively, and they admit small balanced separators:

\textbf{Theorem 2 (Planar Separator Theorem, Lipton and Tarjan [16]).} Every planar graph has a $(2/3, \sqrt{n})$-balanced separator, which can be found in time $O(n)$.

\textbf{Theorem 3 (Bounded Genus Separator Theorem, Gilbert, Hutchinson, and Tarjan [14]).} Every graph of genus $g$ has a $(2/3, \sqrt{gn})$-balanced separator, which can be found in time $O(n)$.

In a DAG $G$, for a vertex $u$ we can compute the number of all paths from $u$ to $v$ for every vertex $v$ in time $O(m + n)$ via a simple application of topological sort: sum the number of paths to $v$’s in-neighbors. We will refer to this algorithm as CountPaths$(G, u)$. Next we show how to extend known single source shortest path (SSSP) algorithms with counting:

Except for the SSSP call, this algorithm runs in linear time and computes the number of shortest paths from $s$ to every vertex in $G$. It does this by building a DAG of tight edges. By Observation 1, since every edge has weight greater than 0, shortest paths are simple. Thus, no cycles can be added into the DAG $G'$ while looping over the edges. For every shortest path $p$ between $u$ and an arbitrary vertex $v \in V(G)$, every edge of $p$ will be added in the direction of the path by Observation 1 (an edge is a two-vertex sub-path of the shortest path $p$). This leads to the following lemma:

\textbf{Lemma 4.} For any SSSP algorithm with running time $T(n)$ there exist an SSSP counting algorithm with running time $T(n) + O(m)$. 
Algorithm 1 Compute #shortest paths from $u$ to every vertex in $G$.

procedure CountShortestPaths($G, u$)
  SSSP($G, u$) [Use an existing algorithm, assume $d[v]$ stores the distance to vertex $v$.]
  initialize unweighted DAG $G' = (V(G), \emptyset)$
  for edge $e = (v, w)$ in $G$
    if $d[w] = d[v] + w(e)$ then
      insert arc $(v, w)$ into $G'$
    else if $d[v] = d[w] + w(e)$ then
      insert arc $(w, v)$ into $G'$
  CountPaths($G', u$)

Of special importance is the application of this approach to planar graphs where Henzinger et al. [15] designed an $O(n)$ SSSP algorithm. Hence, in planar graphs we can count single source shortest paths in $O(n)$ time. The approach of [15] extends to bounded genus graphs, where it gives an $O(h(g)n)$ running time for graphs of genus $g$ (where $h()$ is a function dependent only on $g$).

3 Counting Oracle for All Paths in Planar DAGs

In this section, we prove the following theorem:

▶ Theorem 5. For any planar DAG $G$, there exists an oracle for #PATH-DAG-ORACLE which takes $O(n^{1.5})$ space, takes $O(n^{1.5})$ time to construct, and for any pair of vertices $s, t \in V(G)$ the oracle can answer queries about the number of paths from $s$ to $t$ in $O(\sqrt{n})$ time.

3.1 Building the Oracle

A naive algorithm for counting the number of paths between two vertices in an unweighted DAG takes $O(n^2)$ time by running CountPaths from every possible source vertex. Instead, we induce Theorem 2 to construct the oracle in a divide-and-conquer manner: We first find a separator $(A, B, C)$ for the given graph. Then we count the number of paths that intersect the separating set $C$. Finally, we count the number of paths that lie entirely within sub-graphs induced by $A$ and $B$, respectively. For planar graphs this will lead to an $O(n^{1.5})$ construction time and $O(n^{1.5})$ space. The tricky aspect comes from the fact that many of the paths may cross the separator multiple times.

We start by defining a notion of paths intersecting sets and introduce two sets that are closely related to the oracle algorithm. Then we state a structural relation between these sets, the proof of which we defer to the full version of the paper.

▶ Definition 6. Let $G = (V, E)$ be a graph. We say a path $p$ intersects a set $S \subseteq V$ if $p$ contains a vertex from $S$. A vertex $v$ is a first $S$-intersecting vertex of $p$ if and only if $v \in p \cap S$ and there is no other $u \in p \cap C$ such that $u \prec_p v$.

▶ Definition 7. Let $(A, B, C)$ be a separator of $G$. For a pair of vertices $u$ and $v$, define:
  - $P_G(u, v)$ as the set of simple $u$-$v$ paths in $G$, and
  - $P_{G,C}(u, v, c)$ as the set of simple $u$-$v$ paths in $G$ with first $C$-intersecting vertex $c \in C$.

▶ Lemma 8. For a DAG $G$ with separator $(A, B, C)$ and $c \in C$,
  $$|P_{G,C}(s, t, c)| = |P_{G,C}(s, c, c) \times P_G(c, t)|.$$
Since there are $O(n^2)$ pairs of vertices, we cannot compute $|P_{G,C}(s,c,c) \times P_G(c,t)|$ for every $s,t$ pair in total $O(n^{1.5})$ time. However, if we use the fact that $|S \times T| = |S||T|$ for any two sets $S$ and $T$, we can compute $|P_{G,C}(s,c,c)|$ and $|P_G(c,t)|$ upfront and leave summing over all $c \in C$ to the query time. Using the CountPaths algorithm from Section 2 we can determine $|P_G(c,t)|$ for all $c \in C$ in $O(|C| n)$ time for an unweighted DAG. Since $|C| = O(\sqrt{n})$, we can compute all $|P_G(c,t)|$ in $O(n^{1.5})$ time. It now remains to show that we can compute $|P'_{G,C}(s,c,c)|$ in $O(n^{1.5})$ time.

Directly computing $|P'_{G,C}(s,c,c)|$ will take $O(|A \cup B| n)$ time. This already is $O(n^2)$ and takes too long. To reduce the running time, we instead count paths from the separator. This takes $O(|C| n) = O(n^{1.5})$ time. By reversing the direction of all arcs in the DAG $G$, the number of paths between a pair of vertices is preserved. By modifying $G$, we can guarantee that only $s$-$c$ paths with first $C$-intersecting vertex $c$ remain. We define a new notation for a specific modification of $G$ that is necessary in computing $|P'_{G,C}(s,c,c)|$ efficiently.

\textbf{Definition 9.} Let $G$ be a graph and let $(A,B,C)$ be its separator. For vertex $c \in C$, define $G'_c$ as the graph constructed from $G$ as follows:

- $V(G'_c) = (V(G) \setminus C) \cup \{c\}$, and
- $E(G'_c) = \{(u,v)\mid (v,u) \in E(G) \land u,v \in V(G'_c)\}$.

Intuitively, we remove all vertices from the separator which are not the vertex we are interested in for computing $|P'_{G,C}(s,c,c)|$. Since $s$-$c$ paths with first $C$-intersecting vertex $c$ can only have one vertex in the separator, we remove all paths which could intersect the separator at any other vertex. We also reverse all remaining arcs in the graph. The relationship between $G$ and $G'_c$, which we prove in the full version of the paper, is as follows:

\textbf{Lemma 10.} For a DAG $G$ with separator $(A,B,C)$ and $c \in C$, $|P'_{G,C}(s,c,c)| = |P_{G_c}(s,c)|$.

Lemmas 8 and 10 suggest the following oracle construction algorithm:

\begin{algorithm}
\textbf{procedure} \textit{ConstructAllPathsOracleDAG}\textit{(G)}
\begin{algorithmic}
\IF{$|V(G)| = 0$} \RETURN \ENDIF
\STATE find a separator $(A,B,C)$ in $G$ (and store it)
\FOR{$c \in C$} 
\STATE build $G'_c$ by removing $C \setminus \{c\}$ from $G$ and reversing arcs
\STATE call $\text{CountPaths}(G,c)$ and store the results as $P_G[c,v]$ for every $v \in V(G)$
\STATE call $\text{CountPaths}(G'_c,c)$ and store the results as $P_{G_c}[c,v]$ for every $v \in V(G)$
\STATE $\text{ConstructAllPathsOracleDAG}(G[A])$, where $G[A]$ is the $A$-induced subgraph
\STATE $\text{ConstructAllPathsOracleDAG}(G[B])$
\ENDFOR
\end{algorithmic}
\end{algorithm}

To bound the running time of the construction of the oracle, let $\alpha$ be the constant from the separator definition, used to bound the sizes of the sets $A$ and $B$. By Theorem 2, $\alpha \leq 2/3$ for planar graphs. Then, we get the following recurrence for the running time:

$$T(n) = \begin{cases} T(|A|) + T(|B|) + O(n^{1.5}) \leq T(\alpha n) + T((1-\alpha)n) + O(n^{1.5}) & \text{if } n \geq 1 \\ O(1) & \text{if } n = 0 \end{cases}$$

where the $O(n^{1.5})$ term comes from doing $O(\sqrt{n})$ of the CountPaths computations, and the inequality is a worst case bound which follows from $T$'s convexity. This recurrence can be evaluated using the Akra-Bazzi Method [2]. Trivially, the $p$ value for which $(\sum_i a_i b_i^p = 1)$
We note that in many cases both the running time and the space requirements can be

time of a query is

\[ T(n) = O\left(n^1 \left(1 + \int_1^n \frac{u^{1.5}}{w} du\right)\right) = O(n^{1.5}). \]

We note that in many cases both the running time and the space requirements can be

significantly smaller, for example for graphs with \(O(1)\) size separators such as outerplanar

graphs.

At each recursive call of the oracle construction we store, for each \(c \in C\), both \(P_G[c, v]\) and

\(P_{G'}[c, v]\). This results in \(O(|C|n) = O(n^{1.5})\) space per recursive call, yielding the following

recurrence for the total space needed by the oracle: \(S(n) = S(A) + S(B) + O(n^{1.5})\). This is

exactly the same recurrence as the one for the running time of building the oracle. Thus, the

amount of space needed to store the oracle is \(O(n^{1.5})\).

### 3.2 Querying the Oracle

To query the oracle, we essentially compute \(\sum_{c \in C} P_{G'}[c, s]P_G[c, t]\) for each depth until (and

including) \(s\) and \(t\) become separated by the separator. This can be done using the following

algorithm:

**Algorithm 3** Query the \#PATH-DAG-ORACLE.

```
procedure QueryPaths(G, s, t)
numPaths = 0
while (s, t ∈ A) or (s, t ∈ B), where (A, B, C) is the stored separator of G do
  for c ∈ C do
    numPaths+= \(P_{G'}[c, s]P_G[c, t]\)
    if s, t ∈ A then G = G[A]
    else G = G[B]
  for c ∈ C do
    numPaths+= \(P_{G'}[c, s]P_G[c, t]\)
```

This algorithm relies on Lemmas 8 and 10. For any position of \(s\) and \(t\), the paths from \(s\)

to \(t\) can be split into two groups: those that intersect \(C\) and those that do not. The paths

that intersect \(C\) contribute \(\sum_{c \in C} |P_{G'}(c, s)||P_G(c, t)| = \sum_{c \in C} P_{G'}[s, c]P_G[t, c]\). Notice that

this computation holds also in the case when \(s \in C\) (in which case \(|P_{G'}(c, s)| = 1\) for \(c = s\)

and \(|P_{G'}(c, s)| = 0\) for every other \(c\), or when \(t \in C\) (in which case \(|P_{G'}(c, t)| = 1\) for \(c = t\)

and \(|P_{G'}(c, t)| = 0\) for every other \(c\). The paths that do not intersect \(C\), which occur only

when \(s\) and \(t\) are either both in \(A\) or both in \(B\), are entirely contained within \(G[A]\) or \(G[B]\).

Hence, it suffices to recurse on the respective side of the graph.

It remains to analyze the running time of the query algorithm. Since the oracle has

already been computed, the addition steps each take \(O(1)\) time. The running time of this

algorithm is bounded by the maximum depth before \(s\) and \(t\) are split by a separator and by

the number of vertices in a separator at each depth. Since the separator is balanced, we have

\(T(n) \leq T(\alpha n) + O(\sqrt{n})\). With a simple application of the Master Theorem, the running

time of a query is \(O(\sqrt{n})\). This concludes the proof of Theorem 5.
We have shown that an oracle can be built for planar DAGs for counting the number of paths between any pair of vertices. In this section, we prove the existence of a similar data structure for shortest paths on any planar graph with positive edge weights. As noted in Observation 1, if all edge weights are positive, then shortest paths must be simple. We first prove shortest path versions of Lemmas 8 and 10. For notational convenience, we define two relevant sets of shortest paths:

**Definition 11.** For a positively weighted graph $G = (V, E, w)$ with a separator $(A, B, C)$ and vertices $u$ and $v$, define:
- $Q_G(u, v)$ as the set of shortest paths from $u$ to $v$ in $G$, and
- $Q'_{G,C}(u, v, c)$ as the set of shortest paths from $u$ to $v$ in $G$ with first $C$-intersecting vertex $c \in C$.

Note that the paths in $Q_G(u, v)$ are always simple by Observation 1. However, the paths in $Q'_{G,C}(u, v, c)$ do not have to be simple since they are required to pass through a specific vertex. Next we extend the notion of edge and path lengths to sets:

**Definition 12.** Let $S$ be a set of paths in $G$. We define $w(S)$ as the length of the shortest path in $S$. If $S = \emptyset$, $w(S) = \infty$.

In particular, $w(Q_G(u, v))$ is the length of the shortest $u$-$v$ path in $G$, and $w(Q'_{G,C}(u, v, c))$ is the length of the shortest $u$-$v$ path in $G$ which has a first $C$-intersecting vertex $c$. We make a note that for some vertices $c \in C$, it may be the case that $w(Q'_{G,C}(u, v, c)) > w(Q_G(u, v))$. However, if $s \in A$ and $t \in B$, then there must be some $c \in C$ such that $w(Q'_{G,C}(u, v, c)) = w(Q_G(u, v))$ as any shortest path $p \in Q_G(u, v)$ must intersect separator $C$. There may be multiple such $c$, but at least one must always exist.

With these definitions in place, we now give a similar set of lemmas to compute $|Q_G(u, v)|$ by computing $|Q'_{G,C}(u, v, c)|$ for all paths that intersect the separator $C$. For the cases where $s \in C$ or $t \in C$, we note that the $s$-$s$ and the $t$-$t$ paths consisting of only one vertex have a weight of $0$. As before, the remaining paths which lie entirely within $A$ or $B$ can be counted with recursion.

**Lemma 13.** For a graph $G$ with separator $(A, B, C)$ and a vertex $c \in C$, $|Q'_{G,C}(s, t, c)| = |Q'_{G,C}(s, c, c) \times Q_G(c, t)|$ and $w(Q'_{G,C}(s, t, c)) = w(Q'_{G,C}(s, c, c)) + w(Q_G(c, t))$.

**Proof.** To prove the first part, we show a bijection between $Q'_{G,C}(s, t, c)$ and $Q_G(c, t)$ as follows: let $p_1$ be the $s$-$c$ sub-path of $p$ and $p_2$ be the $c$-$t$ sub-path of $p$. By Observation 1, both $p_1$ and $p_2$ must be shortest paths. Since we split $p$ at $c$, $p_1$ only intersects the separator at $c$ and thus $p_1 \in Q'_{G,C}(s, c, c)$. Since $p_2$ is a shortest path, $p_2 \in Q_G(c, t)$. Also, the map from $p$ to $(p_1, p_2)$ is injective by the same argument as in Lemma 8.

Conversely, let path $p_1 \in Q'_{G,C}(s, c, c)$ and path $p_2 \in Q_G(c, t)$. Since both $p_1$ and $p_2$ are shortest paths, it follows that the path $p$ formed by concatenating $p_1$ and $p_2$ is a shortest path with respect to all $s$-$t$ paths with first $C$-intersecting $c$. If a shorter $s$-$t$ path with $C$-intersecting vertex $c$ existed, then it either contains a shorter $s$-$c$ or $c$-$t$ sub-path than $p_1$ or $p_2$ respectively which contradicts $p_1$ and $p_2$ being shortest paths. Thus for any pair

---

2 This is not true for all $s$-$t$ paths. There may be a shorter $s$-$t$ path with a different first $C$ crossing vertex. Such a case will be determined by a query by comparing path lengths across the vertices in the separator.
\( p_1 \in Q'_{G,C}(s, c, c) \) and \( p_2 \in Q_G(c, t) \), there is a corresponding path \( p \in Q'_{G,C}(s, t, c) \) which maps to \((p_1, p_2)\) by the above map. Thus the map is also surjective.

Since \( p \) was formed by concatenating \( p_1 \) with \( p_2 \), we have \( w(p) = w(p_1) + w(p_2) \). Because all paths in each of \( Q \) and \( Q' \) have the same weight, \( w(Q'_{G,C}(s, t, c)) = w(Q'_{G,C}(s, c, c)) + w(Q_G(c, t)) \).

We next show how to compute \(|Q'_{G,C}(s, c, c)|\). For a graph \( G \) with edge weights \( w_G \), construct \( G'_c \) according to Definition 9 and add weights \( w_{G'_c} \) as follows: for \((u, v) \in E(G'_c)\) let \( w_{G'_c}(u, v) = w_G(v, u) \). We get the following weighted version of Lemma 10:

\[ \text{Lemma 14. In a graph } G \text{ with separator } (A, B, C) \text{ and a vertex } c \in C, |Q'_{G,C}(s, c, c)| = |Q_{G'_c}(c, s)| \text{ and } w(Q'_{G,C}(s, c, c)) = w(Q_{G'_c}(c, s)). \]

The last piece we need in order to build an oracle for the number of shortest paths in a planar graph is a way to compute \(|Q_G(u, v)|\) efficiently. As discussed in Section 2, in planar graphs we can compute both \(|Q_G(u, v)|\) and \(w(Q_G(u, v))\) in \(O(n)\) time for all vertices \(v \in V(G)\) and a source vertex \(u \in V(G)\). Then, we can build an oracle for counting shortest paths by mimicking Algorithm 2 where the \text{COUNTPATHS} calls get replaced with \text{COUNTSHORTESTPATHS} calls, see Algorithm 1. As before, both construction time for the oracle and the space needed are \(O(n^{1.5})\) for planar graphs.

**Algorithm 4** Build a Shortest Path Counting Oracle for graph \( G \).

```
procedure \text{ConstructShortestPathOracle}(G)
if \(|V(G)| = 0\) then return
find a separator \((A, B, C)\) in \( G \) (and store it)
for \( c \in C \) do
  construct graph \( G'_c \) (see Definition 9, plus add weights)
call \text{CountShortestPaths}(G, c)
call \text{CountShortestPaths}(G'_c, c)
store the respective counts as \(Q_G[c, v]\) and \(Q_{G'_c}[c, v]\),
  store also the corresponding path lengths as \(w_G[c, v]\) and \(w_{G'_c}[c, v]\), respectively
\text{ConstructOracle}(G[A])
\text{ConstructOracle}(G[B])
```

Querying the oracle requires a few extra conditions, see Algorithm 5, but it can still be done in time \(O(\sqrt{n})\). As we noted before, for some \(c \in C\), \(w(Q'_{G,C}(s, t, c))\) may be larger than \(w(Q_G(s, t))\). We can detect this by comparing \(w(Q'_{G,C}(s, c, c)) + w(Q'_{c, t, G})\). Since at least one \(c \in C\) must have \(w(Q'_{G,C}(s, t, c)) = w(Q_G(s, t))\), we can have our query determine which \(c\) satisfy \(\min w(Q'_{G,C}(s, c, c)) + w(Q_G(c, t))\) and only add counts from those \(c\). We assume that the distance results from the SSSP are stored along with the number of shortest paths as before. We double the amount of space required, but this still falls within \(O(n^{1.5})\) space.

This analysis leads to the following theorem:

\[ \text{Theorem 15. For any planar graph } G \text{, there exists an oracle for } \#\text{SHORTPATH-ORACLE} \text{ which takes } O(n^{1.5}) \text{ space, takes } O(n^{1.5}) \text{ time to construct, and for any pair of vertices } s, t \in V(G) \text{ the oracle can answer queries about the number of paths from } s \text{ to } t \text{ in } O(\sqrt{n}) \text{ time.} \]
Algorithm 5 Querying the #SHORTPATH-ORACLE.

procedure QUERYPATHS(G, s, t)
numPaths = 0
minDist = \infty
while (s, t \in A) or (s, t \in B), where (A, B, C) is the stored separator of G do
for c \in C do
  if \text{w}_{G'}[c, s] + \text{w}_{G'}[c, t] < minDist then
    minDist = \text{w}_{G'}[c, s] + \text{w}_{G'}[c, t]
    numPaths = Q_{G'}[c, s]Q_{G'}[c, t]
  else if \text{w}_{G'}[c, s] + \text{w}_{G'}[c, t] = minDist then
    numPaths += Q_{G'}[c, s]Q_{G'}[c, t]
if s, t \in A then G = G[A]
else G = G[B]
for c \in C do
  if \text{w}_{G'}[c, s] + \text{w}_{G'}[c, t] < minDist then
    minDist = \text{w}_{G'}[c, s] + \text{w}_{G'}[c, t]
    numPaths = Q_{G'}[c, s]Q_{G'}[c, t]
  else if \text{w}_{G'}[c, s] + \text{w}_{G'}[c, t] = minDist then
    numPaths += Q_{G'}[c, s]Q_{G'}[c, t]

5 Generalizing the Oracle

In this section, we relax the constraints of the previous sections to generalize the oracle data structure. The constraints we require are as follows:

- G has positive edge weights, and
- G has a \((\alpha, f(n))\)-balanced \((A, B, C)\) separator which can be found in time \(O(g(n))\).

Then, we can use Algorithms 1-5 (in fact, Algorithm 1 works for any graph and does not require separators) as stated and the proofs of correctness still hold. However, we need to rework the running time estimates and space bounds. The running time of the oracle construction is given by the following recurrence, where \(T_{SSSP}(n)\) denotes the running time of SSSP:

\[ T(n) \leq T(\alpha n) + T((1 - \alpha)n) + O(T_{SSSP}(n)f(n)) + O(g(n)). \]

For simplicity, let us express the additive term as \(\hat{f}(n)\). As before, this recurrence can be evaluated using the Akra-Bazzi Method. Again, the \(p\) value for which \(\sum_i a_i b_i^p = 1\) is 1. (Take \(a_0 = a_1 = 1, b_0 = \alpha,\) and \(b_1 = 1 - \alpha.\) With \(p = 1\), the recurrence is evaluated as follows:

\[ T(n) = \Theta\left( x^1 \left( 1 + \int_1^x \frac{\hat{f}(u)}{u^2} du \right) \right). \]

This splits nicely into three cases.

1. If \(\hat{f}(n) = o(n)\), then \(T(n) = \Theta(n)\).
2. If \(\hat{f}(n) = \Theta(n \log^n n)\), then \(T(n) = \Theta(n \log^{a+1} n)\).
3. If for every \(a\) we have \(\hat{f}(n) = \omega(n \log^a n)\), then \(T(n) = \Theta(\hat{f}(n))\).
Therefore, the running time of the oracle construction can be determined using \( \hat{f}(n) = O(T_{SSSP}(n)f(n)) + O(g(n)) \). The space requirement of this algorithm is \( f(n) = O(nf(n) + n) \) (this can be done by only storing which side of the separator a vertex lies in \((A, B, \text{ or } C)\), distances and numbers of paths to a vertex for each vertex in the separator.

In real applications of this oracle, case 1 will never occur as running SSSP takes \( \Omega(n) \) time in any graph. However, case 2 may occur. In fact, for outerplanar graphs, which have \( m = O(n) \) (since they are planar) and which also have \( O(1) \) separators, case 2 applies, giving a running time of \( O(n \log n) \) to build this oracle and a running time of \( O(\log n) \) to query it.

The time to query this oracle data structure is given by the following recurrence: \( T(n) = T(\alpha n) + 2f(n) \). As before, we only query \( A \) or \( B \) until the separator splits vertices \( s \) and \( t \). This recurrence can be evaluated with the Master Theorem giving a query time as follows:

1. If \( f(n) = o(\log n) \), then \( T(n) = \Theta(\log n) \).
2. If \( f(n) = \Omega(\log n) \), then \( T(n) = \Theta(f(n)) \).

Putting together all of this, we have the following theorem.

\[ \textbf{Theorem 16.} \text{ Let } G \text{ be a graph with } (\alpha, f(n)) \text{ balanced separators which can be found in } g(n) \text{ time and let } T_{SSSP}(n) \text{ be the time needed to solve SSSP for } G. \text{ Let } f(n) = T_{SSSP}(n)f(n) + g(n). \text{ An oracle data structure for counting the number of shortest paths in } G \text{ between any pair of vertices can be computed in the following time bounds:} \]

\[
\begin{array}{ccc}
\text{Time} & \text{Construction Time} & \text{Query Time} \\
\alpha(n) & T(n) = \Theta(n) & f(n) = o(\log n) \\
\Theta(\log a n) & T(n) = \Theta(n \log a+1 n) & f(n) = \Omega(\log n) \\
\omega(\log a n) & T(n) = \Theta(f(n)) & f(n) = \Theta(\log n)
\end{array}
\]

The space bounds required are \( O(nf(n) + m) \).

For classes of graphs with small separators and fast SSSP algorithms (e.g. planar graphs and graphs of bounded genus) this oracle can improve the running time. We have seen that for planar graphs, the running time is bounded by \( O(n^{1.5}) \) and the query time for \( O(k) \) pairs is given by \( O(\sqrt{n}k) \). In graphs of bounded genus, SSSP can be done in linear time, in particular \( h(g)n \) for some function \( h \) of the genus \( g \) and it is possible to find a balanced separator of size \( O(\sqrt{g}n) \). Thus these graphs have path counting oracles which can be found in \( O(\sqrt{g}n h(g) n) = O(h(g)g^{0.5} n^{1.5}) \) time, take \( O(\sqrt{g}n) = O(g^{0.5} n^{1.5}) \) space, and answer queries in time \( O(\sqrt{g}n) \).

\[ \textbf{Application.} \text{ We conclude with mentioning how our oracle provides an improvement in the running time of the algorithm of Chambers, Fox, and Nayyeri [7] counting minimum } (s,t)-\text{cuts in graphs of bounded genus. Due to space constraints we do not reproduce their algorithm here but only discuss the parts that are relevant to our improvement of their original running time of } 2^{O(g)}n^2. \text{ The main component contributing to this running time (see Section 5.3 in [7]) is iterating through } 2^{O(g)} \text{ "crossing sequences," which determine the different } \text{shapes} \text{ of the possible minimum } (s,t)-\text{cuts. For each such crossing sequence a DAG embedded in a surface of the same genus is constructed, and in this DAG one needs to compute the number of paths between } O(n) \text{ pairs of vertices. Arithmetic operations (addition, multiplication) on these numbers then yield the desired number of minimum } (s,t)-\text{cuts. The original work simply bounded the running time needed for these cut-counts as } O(n^2), \text{ yielding an overall } 2^{O(g)}n^2 \text{ running time. Using our oracle approach, the running time becomes } O(\sqrt{g}n) \text{ per query with } O(n) \text{ queries, totaling } 2^{O(g)}\sqrt{g}n^{1.5} \text{ time, which includes the construction of the } 2^{O(g)} \text{ oracles. The overall improved running time, including a maximum flow computation and a triangulation transformation of the input graph (both of which can be bounded by } O(n^2) \text{), is then } 2^{O(g)}\sqrt{g}n^{1.5} + O(n^2). \]
On Counting Oracles for Path Problems

References

Reconstructing Phylogenetic Tree From Multipartite Quartet System

Hiroshi Hirai
Department of Mathematical Informatics, Graduate School of Information Science and Technology, The University of Tokyo, Japan
hirai@mist.i.u-tokyo.ac.jp

Yuni Iwamasa
Department of Mathematical Informatics, Graduate School of Information Science and Technology, The University of Tokyo, Japan
yuni_iwamasa@mist.i.u-tokyo.ac.jp

Abstract
A phylogenetic tree is a graphical representation of an evolutionary history in a set of taxa in which the leaves correspond to taxa and the non-leaves correspond to speciations. One of important problems in phylogenetic analysis is to assemble a global phylogenetic tree from smaller pieces of phylogenetic trees, particularly, quartet trees. QUARTET Compatibility is to decide whether there is a phylogenetic tree inducing a given collection of quartet trees, and to construct such a phylogenetic tree if it exists. It is known that QUARTET Compatibility is NP-hard but there are only a few results known for polynomial-time solvable subclasses.

In this paper, we introduce two novel classes of quartet systems, called complete multipartite quartet system and full multipartite quartet system, and present polynomial time algorithms for QUARTET Compatibility for these systems. We also see that complete/full multipartite quartet systems naturally arise from a limited situation of block-restricted measurement.

2012 ACM Subject Classification Mathematics of computing → Combinatorial algorithms

Keywords and phrases phylogenetic tree, quartet system, reconstruction

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.57

Acknowledgements We thank Kunihiko Sadakane for bibliographical information.

1 Introduction

A phylogenetic tree for finite set \([n] := \{1, 2, \ldots, n\}\) is a tree \(T = (V, E)\) such that the set of leaves of \(T\) coincides with \([n]\) and each internal node \(V \setminus [n]\) has at least three neighbors. A phylogenetic tree represents an evolutionary history in a set of taxa in which the leaves correspond to taxa and the non-leaves correspond to speciations. One of important problems in phylogenetic analysis is to assemble a global phylogenetic tree on \([n]\) (called a supertree) from smaller pieces of phylogenetic trees on possibly overlapping subsets of \([n]\); see [17, Section 6].

A quartet tree (or quartet) is a smallest nontrivial phylogenetic tree, that is, it has four leaves (as taxa) and it is not a star. There are three quartet trees in set \(\{a, b, c, d\}\), which are denoted by \(ab||cd\), \(ac||bd\), and \(ad||bc\). Here \(ab||cd\) represents the phylogenetic tree such that \(a\) and \(b\) (\(c\) and \(d\)) are adjacent to a common node; see Figure 1. Quartet trees are

1 Supported by KAKENHI Grant Numbers JP26280004, JP17K00029.
2 Supported by JSPS Research Fellowship for Young Scientists.
used for representing substructures of a (possibly large) phylogenetic tree. A fundamental problem in phylogenetic analysis is to construct a phylogenetic tree having given quartets as substructures. To introduce this problem formally, we need some notations and terminologies. We say that a phylogenetic tree \( T \) displays a quartet \( ab|cd \) if the simple paths connecting \( a, b \) and \( c, d \) in \( T \), respectively, do not meet, i.e., \( ab|cd \) is the “restriction” of \( T \) to leaves \( a, b, c, d \); see Figure 2. By a quartet system on \( [n] \) we mean a collection of quartet trees whose leaves are subsets of \( [n] \). We say that \( T \) displays a quartet system \( \mathcal{Q} \) if \( T \) displays all quartet trees in \( \mathcal{Q} \). A quartet system \( \mathcal{Q} \) is said to be compatible if there exists a phylogenetic tree displaying \( \mathcal{Q} \). Now the problem is formulated as:

**Quartet Compatibility**

**Given:** A quartet system \( \mathcal{Q} \).

**Problem:** Determine whether \( \mathcal{Q} \) is compatible or not. If it is compatible, obtain a phylogenetic tree \( T \) displaying \( \mathcal{Q} \).

**Quartet Compatibility** has been intensively studied in computational biology as well as theoretical computer science, particularly, algorithm design and computational complexity. After a fundamental result by Steel [18] on the NP-hardness of **Quartet Compatibility**, there have been a large amount of algorithmic results, e.g., efficient heuristics [13, 19], approximation algorithms [3, 4, 12], and parametrized algorithms [7, 10].

In contrast, there are only a few results known for polynomial-time solvable special subclasses:

- Colonius–Schulze [8] established a complete characterization to the abstract quaternary relation \( N \) (neighbors relation) obtained from a phylogenetic tree \( T \) by: \( N(a, b, c, d) \) holds if and only if \( T \) displays quartet tree \( ab|cd \). By using this result, Bandelt–Dress [2] showed that if, for every 4-element set \( \{a, b, c, d\} \) of \( [n] \), exactly one of \( ab|cd \), \( ac|bd \), and \( ad|cd \) belongs to \( \mathcal{Q} \), then **Quartet Compatibility** for \( \mathcal{Q} \) can be solved in polynomial time.

- Aho–Sagiv–Szymanski–Ullman [1] devised a polynomial time algorithm to find a rooted phylogenetic tree displaying the input triple system. By using this result, Bryant–Steel [5] showed that, if all quartets in \( \mathcal{Q} \) have a common label, then **Quartet Compatibility** for \( \mathcal{Q} \) can be solved in polynomial time.

Such results are useful for designing experiments to obtain quartet information from taxa, and also play key roles in developing supertree methods for (incompatible) phylogenetic trees (e.g., [16]).

In this paper, we present two novel tractable classes of quartet systems. To describe our result, we extend the notions of quartets and quartet systems. In addition to \( ab|cd \), we consider symbol \( ab|cd \) as a quartet, which represents the quartet tree \( ab|cd \) or the star with leaves \( a, b, c, d \); see Figure 1. This corresponds to the weak neighbors relation in [2, 8], and enables us to capture a degenerate phylogenetic tree in which internal nodes may have degree greater than 3. In a sense, \( ab|cd \) means a “possibly degenerate” quartet tree such that the center edge can have zero length. We define that a phylogenetic tree \( T \) displays \( ab|cd \) if the simple paths connecting \( a, b \) and \( c, d \) in \( T \), respectively, meet at most one node, i.e., the restriction of \( T \) to \( a, b, c, d \) is \( ab|cd \) or star; see Figure 2. Then the concepts of quartet systems, displaying, compatibility, and **Quartet Compatibility** are naturally extended. A quartet system \( \mathcal{Q} \) is said to be full on \( [n] \) if, for each distinct \( a, b, c, d \in [n] \), either one of \( ab|cd, ac|bd, ad|bc \) belongs to \( \mathcal{Q} \) or all \( ab|cd, ac|bd, ad|bc \) belong to \( \mathcal{Q} \). The latter situation says that any phylogenetic tree displaying \( \mathcal{Q} \) should induce a star on \( a, b, c, d \). Actually the above polynomial-time algorithm by Bandelt–Dress [2] works for full quartet systems.
Figure 1 The quartets \(ab|cd, ac|bd\), and \(ad|bc\) represent the first, second, and third phylogenetic trees for \(a, b, c, d\) from the left, respectively. \(ad|bc\), for example, represents one of the two phylogenetic trees in the dotted curve, that is, \(ad|bc\) or the star graph with leaves \(a, b, c, d\).

Figure 2 An example of phylogenetic tree \(T\) for \(\{1, 2, \ldots, 9\}\). \(T\) displays, for example, 13||79, 13|79, and 13|46, 14|36, 16|34.

Full quartet systems may be viewed as a counter part of complete graphs. We introduce multipartite counterparts for quartet systems. A quartet system \(Q\) is said to be complete bipartite relative to bipartition \(\{A, B\}\) of \([n]\) with \(\min\{|A|, |B|\} \geq 2\) if, for all distinct \(a, a' \in A\) and \(b, b' \in B\), \(Q\) has exactly one of

\[
ab|a'b', \quad ab'|a'b', \quad aa'|bb',
\]

and every quartet in \(Q\) is of the above form (1). Note that every phylogenetic tree displays exactly one of three quartets in (1). We next introduce a complete multipartite system. Let \(\mathcal{A} := \{A_1, A_2, \ldots, A_r\}\) be a partition of \([n]\) with \(|A_i| \geq 2\) for all \(i \in [r]\). A quartet system \(Q\) is said to be complete multipartite relative to \(\mathcal{A}\) or complete \(\mathcal{A}\)-partite if \(Q\) is represented as \(\bigcup_{1 \leq i < j \leq r} Q_{ij}\), for complete bipartite quartet systems \(Q_{ij}\), on \(A_i \cup A_j\) with bipartition \(\{A_i, A_j\}\). A quartet system \(Q\) is said to be full multipartite relative to \(\mathcal{A}\) or full \(\mathcal{A}\)-partite if \(Q\) is represented as \(Q_0 \cup Q_1 \cup \cdots \cup Q_r\), where \(Q_0\) is a complete \(A_1\)-partite quartet system and \(Q_i\) is a full quartet system on \(A_i\) for each \(i \in [r]\). Our main result is:

**Theorem 1.1.** If the input quartet system \(Q\) is complete \(\mathcal{A}\)-partite or full \(\mathcal{A}\)-partite, then quartet compatibility can be solved in \(O(|\mathcal{A}|n^2)\) time.

The result for full \(\mathcal{A}\)-partite quartet systems extends the above polynomial time solvability for full quartet systems by [2]. Also this result has some insights on supertree construction from phylogenetic trees on disjoint groups of taxa. In such a case, we have a full system on each group. Another possible application is given as follows.
Reconstructing Phylogenetic Tree From Multipartite Quartet System

Application: Inferring a phylogenetic tree from block-restricted measurements. Quartet-based phylogenetic tree reconstruction methods may be viewed as qualitative approximations of distance methods that construct a phylogenetic tree from (evolutionary) distance \( \delta : [n] \times [n] \rightarrow [0, \infty] \) among a set \([n]\) of taxa. Here \( [n] \) denotes the set of nonnegative real values. The distance \( \delta \) naturally gives rise to a full quartet system \( Q \) as follows. Let \( Q := \emptyset \) at first. For all distinct \( a, b, c, d \in [n] \), add \( ab \mid cd \) to \( Q \) if \( \delta(a, b) + \delta(c, d) \leq \min\{\delta(a, c) + \delta(b, d), \delta(a, d) + \delta(b, c)\} \).

See [9, 14]. Then \( Q \) becomes a full quartet system, after adding \( ab \mid cd, ac \mid bd, ad \mid bc \) if none of \( ab \mid cd, ac \mid bd, ad \mid bc \) belong to \( Q \). If \( \delta \) coincides with the path-metric of an actual phylogenetic tree \( T \) (with nonnegative edge-length), then \( \delta \) obeys the famous four-point condition on all four elements \( a, b, c, d \) [6]:

(4pt) the larger two of \( \delta(a, b) + \delta(c, d), \delta(a, c) + \delta(b, d) \), and \( \delta(a, d) + \delta(b, c) \) are equal.

In this case, the above definition of quartets matches the neighbors relation of \( T \). Thus, from the full quartet system \( Q \), via the algorithm of [2], we can recover the original phylogenetic tree \( T \) (without edge-length).

Next we consider the following limited situation in which complete/full \( A \)-partite quartet systems naturally arise. The set \([n]\) of taxa is divided into \( r \) groups \( A_1, A_2, \ldots, A_r \) (with \( |A_i| \geq 2 \)). By reasons of the cost and/or the difficulty of experiments, we are limited to measure the distance between \( a \in A_i \) and \( b \in A_j \) via different methods/equipments depending on \( i, j \). Namely we have \( \binom{r}{2} \) distance functions \( \delta_{ij} : A_i \times A_j \rightarrow [0, \infty] \) for \( 1 \leq i < j \leq r \) but it is meaningless to compare numerical values of \( \delta_{ij} \) and \( \delta_{i'j'} \) for \( \{i, j\} \neq \{i', j'\} \). A complete \( A \)-partite quartet system \( Q \) is obtained as follows. For distinct \( i, j \), define complete bipartite quartet system \( Q_{ij} \) by: for all distinct \( a, a' \in A_i \) and \( b, b' \in A_j \) it holds

\[
ab \mid a'b' \in Q_{ij} \quad \text{if} \quad \delta_{ij}(a, b) + \delta_{ij}(a', b') < \delta_{ij}(a, b') + \delta_{ij}(a', b),
\]

\[
ab' \mid a'b \in Q_{ij} \quad \text{if} \quad \delta_{ij}(a, b) + \delta_{ij}(a', b') > \delta_{ij}(a, b') + \delta_{ij}(a', b),
\]

\[
aa' \mid bb' \in Q_{ij} \quad \text{if} \quad \delta_{ij}(a, b) + \delta_{ij}(a', b') = \delta_{ij}(a, b') + \delta_{ij}(a', b).
\]

Then \( Q := \bigcup_{1 \leq i < j \leq r} Q_{ij} \) is a complete \( A \)-partite quartet system.

This construction of complete \( A \)-partite quartet system \( Q \) is justified as follows. Assume a phylogenetic tree \( T \) on \([n]\) with path-metric \( \delta \). Assume further that each \( \delta_{ij} \) is linear on \( \delta \), i.e., \( \delta_{ij} \) equal to \( \alpha_{ij} \delta \) for some unknown constant \( \alpha_{ij} > 0 \). By (4pt), the situation \( \delta_{ij}(a, b) + \delta_{ij}(a', b') < \delta_{ij}(a, b') + \delta_{ij}(a', b) \) implies \( \delta(a, b) + \delta(a', b') < \delta(a, b') + \delta(a', b) = \delta(a, a') + \delta(a, b') \), and that \( T \) displays \( ab \mid a'b' \). The situation \( \delta_{ij}(a, b) + \delta_{ij}(a', b') = \delta_{ij}(a, b') + \delta_{ij}(a', b) \) implies \( \delta(a, b) + \delta(a', b') = \delta(a, b') + \delta(a', b) \geq \delta(a, a') + \delta(a, b') \), and that \( T \) displays \( aa' \mid bb' \). Thus, by our algorithm, we can construct a phylogenetic tree \( T' \) “similar” to \( T \) in the sense that \( T' \) and \( T \) produce the same result under our limited measurement.

Suppose now that we have additional \( r \) distance functions \( \delta_i : A_i \times A_i \rightarrow [0, \infty] \) for \( i \in [r] \). In this case, we naturally obtain a full \( A \)-partite quartet system. Indeed, define full quartet system \( Q_i \) on \( A_i \) according to \( \delta_i \) as in the first paragraph. Then \( Q := \bigcup_{1 \leq i < j \leq r} Q_{ij} \cup \bigcup_{1 \leq i \leq r} Q_i \) is a full \( A \)-partite quartet system to which our algorithm is applicable.

Organization. QUARTET COMPATIBILITY can be viewed as a problem of finding an appropriate laminar family. We first introduce a displaying concept for an arbitrary family of subsets, and then divide QUARTET COMPATIBILITY into two subproblems: The first is to find a family displaying the input quartet system, and the second is to transform the family into a desired laminar family. For the second, we utilize the laminarization algorithm developed by Hirai–Iwamasa–Murota–Zivný [11] for a completely irrelevant problem in discrete optimization. In Sections 2 and 3, we show the result for complete and full multipartite quartet systems, respectively. The omitted proofs will be given in the full version of this paper.
Preliminaries. A family $L \subseteq 2^{[n]}$ is said to be laminar if $X \subseteq Y$, $X \supseteq Y$, or $X \cap Y = \emptyset$ holds for all $X, Y \in L$. A phylogenetic tree can be encoded into a laminar family as follows. Let $T = (V, E)$ be a phylogenetic tree for $[n]$. By deleting internal edge $e \in E$, the tree $T$ is separated into two connected components, and so is $[n]$. We denote by $\{X_e, Y_e\}$ the bipartition induced by $e$. By choosing either $X_e$ or $Y_e$ appropriately for each internal edge $e \in E$, we can construct a laminar family $L$ on $[n]$ with $\min\{|X|, |[n] \setminus X|\} \geq 2$ for all $X \in L$. Conversely, let $L$ on $[n]$ be a laminar family with $\min\{|X|, |[n] \setminus X|\} \geq 2$ for all $X \in L$. Then we construct the set $\hat{L} := \{\{X, [n] \setminus X\} \mid X \in L\}$ of bipartitions from $L$. It is known [6] that, for such $\hat{L}$, there uniquely exists a phylogenetic tree that induces $\hat{L}$.

2 Complete multipartite quartet system

2.1 Displaying and Laminarization

In this subsection, we explain that QUARTET COMPATIBILITY for complete multipartite quartet systems can be divided into two subproblems named as DISPLAYING and LAMINARIZATION. Let $A := \{A_1, A_2, \ldots, A_r\}$ be a partition of $[n]$ with $|A_i| \geq 2$ for all $i \in [r]$, and $Q$ be a complete $A$-partite quartet system. We say that a family $F \subseteq 2^{[n]}$ displays $Q$ if, for all distinct $i, j \in [r]$, $a, a' \in A_i$, and $b, b' \in A_j$,

$$ab||a'b' \in Q \iff \text{there is } X \in F \text{ satisfying } a, b \in X \not\equiv a', b' \text{ or } a, b \not\in X \equiv a', b'.$$

We can easily see that, if $L$ is laminar, then $L$ displays exactly one complete $A$-partite quartet system $Q$. Furthermore, such $Q$ is the same as the one displayed by the phylogenetic tree corresponding to $L$. Thus QUARTET COMPATIBILITY for a complete $A$-partite quartet system $Q$ can be viewed as the problem finding a laminar family $L$ displaying $Q$ if it exists.

It can happen that different families may display the same complete $A$-partite quartet system. To cope with such complications, we define an equivalence relation $\sim$ on sets $X, Y \subseteq [n]$ by: $X \sim Y$ if $\{X\}$ and $\{Y\}$ display the same complete $A$-partite quartet system. Let $[X] := \{Y \subseteq [n] \mid X \sim Y\}$ for $X \subseteq [n]$. A set $X \subseteq [n]$ is called an $A$-cut if $X \not\equiv \emptyset$, i.e., $X \not\in \emptyset$. For $X \subseteq [n]$, define

$$\langle X \rangle := \bigcup\{A_i \in A \mid \emptyset \neq X \cap A_i \neq A_i\}. \quad (2)$$

One can see that $X$ is an $A$-cut if and only if $\emptyset \neq X \cap A_i \neq A_i$ holds for at least two $i \in [r]$, i.e., $\langle X \rangle \supseteq A_i \cup A_j$ for some distinct $i, j \in [r]$. We consider only $A$-cuts if the input quartet system $Q$ is complete $A$-partite. Indeed, let $F$ be a family and $F'$ the $A$-cut family in $F$. Then both $F$ and $F'$ display the same complete $A$-partite quartet system.

One can see that, for $A$-cuts $X, Y$, it holds that $X \sim Y \Rightarrow \{\langle X \rangle \cap X, \langle X \rangle \setminus X\} = \{\langle Y \rangle \cap Y, \langle Y \rangle \setminus Y\}$. The equivalence relation is naturally extended to $A$-cut families $F, G$ by: $F \sim G \iff F/\sim = G/\sim$, where $F/\sim := \{\langle X \rangle \mid X \in F\}$. It is clear, by the definition of $\sim$, that if $F \sim G$ then both $F$ and $G$ display the same complete $A$-partite quartet system. An $A$-cut family $F$ is said to be laminarizable if there is a laminar family $L$ with $F \sim L$.

By the above argument, QUARTET COMPATIBILITY for a complete $A$-partite quartet system $Q$ can be divided into the following two subproblems: (i) if $Q$ is compatible, then find a laminarizable family $F$ displaying $Q$, and (ii) if $F$ is laminarizable, then find a laminar family $L$ with $L \sim F$. (i) and (ii) can be formulated as DISPLAYING and LAMINARIZATION, respectively.
Displaying

Given: A complete $\mathcal{A}$-partite quartet system $\mathcal{Q}$.

Problem: Either detect the incompatibility of $\mathcal{Q}$, or obtain some $\mathcal{A}$-cut family $\mathcal{F}$ displaying $\mathcal{Q}$. In addition, if $\mathcal{Q}$ is compatible, then $\mathcal{F}$ should be laminarizable.

Laminarization

Given: An $\mathcal{A}$-cut family $\mathcal{F}$.

Problem: Determine whether $\mathcal{F}$ is laminarizable or not. If $\mathcal{F}$ is laminarizable, obtain a laminar $\mathcal{A}$-cut family $\mathcal{L}$ with $\mathcal{L} \sim \mathcal{F}$.

Here, in LAMINARIZATION, we assume that no distinct $X,Y$ with $X \sim Y$ are contained in $\mathcal{F}$, i.e., $|\mathcal{F}| = |\mathcal{F}/\sim|$.

QUARTET COMPATIBILITY for complete multipartite quartet systems can be solved as follows.

Suppose that $\mathcal{Q}$ is compatible. First, by solving Displaying, we obtain a laminarizable $\mathcal{A}$-cut family $\mathcal{F}$ displaying $\mathcal{Q}$. Then, by solving LAMINARIZATION for $\mathcal{F}$, we obtain a laminar $\mathcal{A}$-cut family $\mathcal{L}$ with $\mathcal{L} \sim \mathcal{F}$. Since $\mathcal{L} \sim \mathcal{F}$, $\mathcal{L}$ also displays $\mathcal{Q}$.

Suppose that $\mathcal{Q}$ is not compatible. By solving Displaying, we can detect the incompatibility of $\mathcal{Q}$ or we obtain some $\mathcal{A}$-cut family $\mathcal{F}$ displaying $\mathcal{Q}$. In the former case, we are done. In the latter case, by solving LAMINARIZATION for $\mathcal{F}$, we can detect the non-laminarizability of $\mathcal{F}$, which implies the incompatibility of $\mathcal{Q}$.

In [11], the authors presented an $O(n^4)$-time algorithm for LAMINARIZATION.

► Theorem 2.1 ([11]). LAMINARIZATION can be solved in $O(n^4)$ time.

In Section 2.3, we give an $O(rn^4)$-time algorithm for Displaying (Theorem 2.8). Thus, by Theorems 2.1 and 2.8, we obtain Theorem 1.1 for complete $\mathcal{A}$-partite quartet systems.

2.2 Algorithm for complete bipartite quartet system

We first construct a polynomial time algorithm for QUARTET COMPATIBILITY for complete bipartite quartet systems. In the following, $\mathcal{A}$ is a bipartition of $[n]$ represented as $\{A,B\}$ with $\min(|A|,|B|) \geq 2$. Note that $X$ is an $\mathcal{A}$-cut if and only if $\emptyset \neq X \cap A \neq A$ and $\emptyset \neq X \cap B \neq B$, and that $X \sim Y$ if and only if $X = Y$ or $X = [n] \setminus Y$.

Choose an arbitrary $a \in [n]$. For a compatible bipartite quartet system $\mathcal{Q}$, there is a laminar $\mathcal{A}$-cut family $\mathcal{F}$ displaying $\mathcal{Q}$ such that there is no $X \in \mathcal{F}$ with $a \in X$. The following proposition implies that such $\mathcal{F}$ is unique.

► Proposition 2.2. Suppose that a bipartite quartet system $\mathcal{Q}$ is compatible. Then a laminarizable $\mathcal{A}$-cut family $\mathcal{F}$ displaying $\mathcal{Q}$ is uniquely determined up to $\sim$.

We introduce two notations used in Sections 2.2.1 and 2.2.2. For $\mathcal{F} \subseteq 2^{[n]}$ and $X \subseteq [n]$, we denote $\{F \cup X \mid F \in \mathcal{F}\}$ by $\mathcal{F} \cup X$. For $C \subseteq A$ and $D \subseteq B$, we denote by $\mathcal{Q}_{C,D}$ the set of quartet trees for $c,c',d,d' (c,c' \in C$ and $d,d' \in D)$ in $\mathcal{Q}$.

2.2.1 Case of $|A| = 2$ or $|B| = 2$

We consider the case of $|A| = 2$ or $|B| = 2$. Without loss of generality, we assume $A = \{a_0,a\}$ with $a_0 \neq a$.

We first explain the idea behind our algorithm (Algorithm 1). Assume that a complete $(\{a_0,a\},B)$-partite quartet system $\mathcal{Q}$ is compatible. By Proposition 2.2, there uniquely exists a laminar $(\{a_0,a\},B)$-cut family $\mathcal{F}$ displaying $\mathcal{Q}$ such that no $X \in \mathcal{F}$ contains $a_0$. 
We consider general complete bipartite quartet systems. The tripartition of a complete bipartite quartet system $\{a_0, a\}$, $B$-partite quartet system with pivot $a$. Then, $\exists b \in B$ such that $b \notin B_k$. Let $F$ be the output families of Algorithm 1 for $\{a_0, a\}$. Let $F$ be the output families of Algorithm 1 for $\{a_0, a\}$ with pivot $a$.

**Algorithm 1** (for complete $\{a_0, a\}$-partite quartet system with pivot $a$).

**Input:** A complete $\{a_0, a\}$-partite quartet system $Q$.

**Output:** Either detect the incompatibility of $Q$, or obtain the (unique) laminar $\{a_0, a\}$-cut family $F$ displaying $Q$ such that no $X \in F$ contains $a_0$.

**Step 1:** If $Q = \emptyset$, that is, $|B|$ is at most one, then output the emptyset and stop.

**Step 2:** Choose an arbitrary $b \in B$. Define $B^+, B^-$, and $B^+$ as (3), (4), and (5), respectively.

**Step 3:** If Algorithm 1 for $\{a_0, a\}$ with pivot $a$ detects the incompatibility of $Q$, then output “$Q$ is not compatible” and stop. Otherwise, let $F^+$ be the output families of Algorithm 1 for $\{a_0, a\}$, $B^+$ and for $\{a_0, a\}$, $B^-$. Define $F = F^- \cup (F^+ \cup (B^- \cup B^-)) \cup ((\{B^+, B^- \cup B^-\} \setminus \emptyset, B)) \cup \{a\}$.

**Step 4:** If $F$ displays $Q$, then output $F$. Otherwise, output “$Q$ is not compatible.”

**Proposition 2.3.** Algorithm 1 solves QUARTET COMPATIBILITY for a complete $\{a_0, a\}$-partite quartet system $Q$ in $O(|Q|)$ time.

### 2.2.2 General case

We consider general complete bipartite quartet systems; $A$ is a bipartition $\{A, B\}$ of $[n]$. As in Section 2.2.1, we first explain the idea behind our algorithm (Algorithm 2). Assume that a complete $A$-partite quartet system $Q$ is compatible. By Proposition 2.2, there uniquely exists a laminar $A$-cut family $F$ displaying $Q$ such that no $X \in F$ contains $a_0$.

Define $F^{a}$ as the output of Algorithm 1 for $\{a_0, a\}$ with pivot $a$. Since $\exists b \in B$ such that $b \notin B_k$. Let $F$ be the output families of Algorithm 1 for $\{a_0, a\}$ with pivot $a$.

Define $F^{a}$ as the output of Algorithm 1 for $\{a_0, a\}$ with pivot $a$. Since $\exists b \in B$ such that $b \notin B_k$. Let $F$ be the output families of Algorithm 1 for $\{a_0, a\}$ with pivot $a$.

It can be easily seen that $F \cap B = \bigcup_{a \in A \setminus \{a_0\}} \{X \cap B \mid X \in F^{a}\}$. In the following, we consider to combine $F^{a}$s appropriately.
Take any $D \in \mathcal{F} \cap B$, and define $A_D := \{a \in A \setminus \{a_0\} \mid \{a\} \cup D \in \mathcal{F}^a\}$. By the laminarity of $\mathcal{F}$, $A_D \cup D$ is the unique maximal set $X$ in $\mathcal{F}$ such that $X \cap B = D$. Hence we can construct the set $G := \{A_D \cup D \mid D \in \mathcal{F} \cap B\} \subseteq \mathcal{F}$ from $\mathcal{F}^a$ ($a \in A \setminus \{a_0\}$). Note that $G$ is laminar.

All the left is to determine all nonmaximal sets $X \in \mathcal{F}$ with $X \cap B = D$ for each $D \in \mathcal{F} \cap B$. Fix an arbitrary $D \in \mathcal{F} \cap B$. Observe that, by the laminarity of $\mathcal{F}$, the set $\{X \in \mathcal{F} \mid X \cap B = D\}$ is a chain $\{X_1, X_2, \ldots, X_m\}$ with $X_1 \subseteq X_2 \subseteq \cdots \subseteq X_m = A_D \cup D$. We are going to identify this chain with the help of Algorithm 1. Let $X^- := \bigcup\{X' \in G \mid X' \subseteq X_m\}$, and choose an arbitrary $b_0 \in B \setminus D$ and $b \in D$. Note that $X_1 \supseteq X^-$ by the laminarity of $\mathcal{F}$. We first consider the easier case $X_1 \cap A \supseteq X^- \cap A$. Then apply Algorithm 1 to $Q|_{A_D \setminus X^- \setminus \{b_0, b\}}$ and obtain $\{(X_1 \setminus X^-) \cap A, (X_2 \setminus X^-) \cap A, \ldots, (X_m \setminus X^-) \cap A\} \cup \{b_0, b\}$ (that displays $Q|_{A_D \setminus X^- \setminus \{b_0, b\}}$). From this we obtain $\{X_1, X_2, \ldots, X_m\}$, as required.

Next consider the case $X_1 \cap A = X^- \cap A$. In this case, by applying Algorithm 1 to $Q|_{A_D \setminus X^- \setminus \{b_0, b\}}$, we only obtain $\{(X_2 \setminus X^-) \cap A, \ldots, (X_m \setminus X^-) \cap A\} \cup \{b_0, b\}$, and hence $\{X_2, \ldots, X_m\}$. Therefore we need to construct $X_1$ individually as follows. Pick any $a \in X^- \cap A$ and retake $b$ from $D \setminus X'$ for maximal $X' \in G$ with $a \in X' \subseteq X^-$. For $a' \in (X_m \setminus X^-) \cap A$, it cannot happen that $ab_0|a'b \in Q$ since all $X \in \mathcal{F}$ containing $a$, $b$ also include $a$. Furthermore we can say that $ab|a'b_0 \in Q$ if and only if $a' \notin X_1(\supset a, b)$. This implies that $ab|a'b_0 \in Q$ if and only if $a'$ belongs to $X_1$. Hence it holds that $X_1$ is the union of $X^- \cup D$ and all elements $a' \in A_D \setminus X^-$ with $aa'|b_0 \in G$.

The formal description of Algorithm 2 is the following; note that, if $F$ is laminar, then $|F|$ is at most $2n$ (see e.g., [15, Theorem 3.5]).

**Proposition 2.4.** Algorithm 2 solves QUARTET COMPATIBILITY for a complete bipartite quartet system $Q$ in $O(|Q|)$ time.

### 2.3 Algorithm for complete multipartite quartet system

In this subsection, we present a polynomial time algorithm for complete multipartite quartet systems. First we introduce some notations before giving the outline of our proposed algorithm (Algorithm 4). Let $A := \{A_1, A_2, \ldots, A_r\}$ be a partition of $[n]$ with $|A_i| \geq 2$ for all $i \in [r]$. For the analysis of the running-time of Algorithm 4, we assume $|A_1| \geq |A_2| \geq \cdots \geq |A_r|$. For $R \subseteq [r]$ with $|R| \geq 2$, let $A_{R} := \{A_i\}_{i \in R}$ and $A_{R} := \bigcup_{i \in R} A_{i}$. For complete $A$-partite quartet system $Q = \bigcup_{1 \leq i < j < r} Q_{ij}$, define $Q_{R} := \bigcup_{1 \leq i < j < r} Q_{ij}$. That is, $Q_{R}$ is the complete $A_{R}$-partite quartet system included in $Q$. For $A_{R}$-cut family $\mathcal{F}$, define $\mathcal{F}_{R} := \{X \cap A_{R} \mid X \in \mathcal{F} \text{ such that } X \cap A_{R} \text{ is an } A_{R}\text{-cut}\}$. Note that $\mathcal{F}_{R}$ is an $A_{R}$-cut family. Then we can easily see the following lemma, which says that partial information $\mathcal{F}_{R}$ of $\mathcal{F}$ can be obtained from $Q_{R}$.

**Lemma 2.5.** Suppose $R \subseteq [r]$ with $|R| \geq 2$. If $Q$ is displayed by $\mathcal{F}$, then $Q_{R}$ is displayed by $\mathcal{F}_{R}$. Furthermore, if $Q$ is compatible, then so is $Q_{R}$.

Our algorithm for DISPLAYING is to construct an $A_{[t]}$-cut family $\mathcal{F}_{t}$ displaying $Q_{[t]}$ for $t = 2, 3, \ldots, r$ in turn as follows.

- First we obtain an $A_{\{1, 2\}}$-cut family $\mathcal{F}_{2}$ displaying $Q_{12}$ by Algorithm 2.
- For $t \geq 2$, we can extend an $A_{[t-1]}$-cut family $\mathcal{F}_{t-1}$ displaying $Q_{[t-1]}$ to an $A_{[t]}$-cut family $\mathcal{F}_{t}$ displaying $Q_{[t]}$ by Algorithm 3. In order to construct $\mathcal{F}_{t}$ in Algorithm 3, we use an $A_{\{t, t\}}$-cut family $\mathcal{G}_{t}$ displaying $Q_{tt}$ for all $i \in [t-1]$. These $\mathcal{G}_{i}$ can be obtained by Algorithm 2.
- We perform the above extension step for $t = 3$ to $t = r$, and then obtain a desired $A$-cut family $\mathcal{F} := \mathcal{F}_{r}$. This is described in Algorithm 4.
Algorithm 2 (for complete bipartite quartet system).
Input: A complete bipartite quartet system $Q$.
Output: Either detect the incompatibility of $Q$, or obtain a laminar $A$-cut family $F$ displaying $Q$.
Step 1: Fix an arbitrary $a_0 \in A$. For each $a \in A \setminus \{a_0\}$, we execute Algorithm 1 for $Q|_{\{a_0,a\},B}$ with pivot $a$. If Algorithm 1 outputs "$Q|_{\{a_0,a\},B}$ is not compatible" for some $a$, then output "$Q$ is not compatible" and stop. Otherwise, obtain the output $F_a$ for each $a$.
Step 2: Let $G := \emptyset$. For each $a \in A \setminus \{a_0\}$, update $G$ as
\[
G \leftarrow \{X \in F_a \mid \exists Y \in G \text{ such that } X \cap B = Y \cap B\}
\cup \{Y \in G \mid \exists X \in F_a \text{ such that } X \cap B = Y \cap B\}
\cup \{\{Y \in G \mid \exists X \in F_a \text{ such that } X \cap B = Y \cap B\} \cup \{a\}\}.
\]
If $|G| > 2n$ for some $a$, then output "$Q$ is not compatible" and stop.
Step 3: If $G$ is not laminar, then output "$Q$ is not compatible" and stop. Otherwise, define $F := G$. For each $X \in G$, do the following:
3-1: Let $X^− := \{X' \in G \mid X' \subseteq X\}$, and choose an arbitrary $b_0 \in B \setminus X$ and $b \in B \cap X$.
3-2: Execute Algorithm 1 for $Q|_{(X \setminus X^−) \cap A, (b_0,b)}$ with pivot $b$. If Algorithm 1 outputs "$Q|_{(X \setminus X^−) \cap A, (b_0,b)}$ is not compatible," then output "$Q$ is not compatible" and stop. Otherwise, define $H :=$ the output family of Algorithm 1 $\cup (X^− \cup (X \cap B))$.
If $X^− \neq \emptyset$, then go to Step 3-3. Otherwise, go to Step 3-4
3-3: Choose an arbitrary $a \in X^− \cap A$ and retake $b$ from $(X \setminus X^−) \cap B$ for maximal $X' \in G$ with $a \in X' \subseteq X^−$. Define $X_1 := X^− \cup (X \cap B) \cup \{a' \in (X \setminus X^−) \cap A \mid aa' \in b_0 b \in Q\}$. If $X_1$ is not included in the minimal element in $H$, then output "$Q$ is not compatible" and stop. Otherwise, update $H \leftarrow H \cup \{X_1\}$.
3-4: $F \leftarrow F \cup H$.
Step 4: If $F$ displays $Q$, then output $F$. Otherwise, output "$Q$ is not compatible."

As a compatible complete bipartite quartet system (Proposition 2.2), a compatible complete multipartite quartet system $Q$ induces some kind of uniqueness of a laminarizable family displaying $Q$, which ensures the validity of our proposed algorithm.

**Proposition 2.6.** Suppose that a complete $A$-partite quartet system $Q$ is compatible. Then a minimal laminarizable $A$-cut family $F$ displaying $Q$ is uniquely determined up to $\sim$.

Algorithm 3 constructs a minimal laminarizable family $F_t$ displaying $Q_{[t]}$ from a minimal laminarizable family $F_{t-1}$ displaying $Q_{[t-1]}$. We define a partial order relation $\prec$ in $A$-cuts by: $X \prec Y$ if $\langle X \rangle \subseteq \langle Y \rangle$ and $\{\langle X \rangle \cap X, \langle X \rangle \setminus X\} = \{\langle X \rangle \cap Y, \langle X \rangle \setminus Y\}$. Define $X \leq Y$ by $X \sim Y$ or $X = Y$. For nonempty $R \subseteq [r]$, we define $\sim_R$ for $A$-cuts by:
\[
X \sim_R Y \iff \{\langle X \rangle \cap X, \langle X \rangle \setminus X\} = \{\langle Y \rangle \cap Y, \langle Y \rangle \setminus Y\},
\]
where $\langle X \rangle := \langle X \rangle \cap A_R$ and $\langle Y \rangle := \langle X \rangle \cap A_R$; recall (2) for the notation $\langle X \rangle$. We abbreviate $\{i_1,i_2,\ldots,i_k\}$ as $i_1i_2\cdots i_k$ for distinct $i_1,i_2,\ldots,i_k$. It is noted that, if $F$ is laminarizable and $X \not\sim Y$ for all distinct $X,Y \in F$, then $|F|$ is at most $2n = 2|A_{[r]}|$. The following proposition shows that Algorithm 3 actually works.
Reconstructing Phylogenetic Tree From Multipartite Quartet System

Algorithm 3 (for extending $\mathcal{F}'$ to $\mathcal{F}$).

**Input:** An $\mathcal{A}$-cut family $\mathcal{F}'$ with $|\mathcal{F}'| \leq 2|\mathcal{A}|_{r-1}$ displaying $\mathcal{Q}_{|r-1|}$.

**Output:** Either detect the incompatibility of $\mathcal{Q}$, or obtain $\mathcal{A}$-cut family $\mathcal{F}$ with $|\mathcal{F}| \leq 2n - 2|\mathcal{A}|_{r-1}$ displaying $\mathcal{Q}$.

**Step 1:** For each $i \in [r-1]$, execute Algorithm 2 for $\mathcal{Q}_{ir}$. If Algorithm 2 returns “$\mathcal{Q}_{ir}$ is not compatible” for some $i \in [r-1]$, then output “$\mathcal{Q}$ is not compatible” and stop. Otherwise, obtain $\mathcal{G}_i$ for all $i \in [r-1]$. Let $\mathcal{F} := \emptyset$.

**Step 2:** If $\mathcal{F}' = \emptyset$, update as $\mathcal{F} := \mathcal{F}' \cup \bigcup_{i \in [r-1]} \mathcal{G}_i$, and go to Step 3. Otherwise, do the following: Take any $X' \in \mathcal{F}'$. Let $\{i_1, i_2, \ldots, i_k\}$ be the set of indices $i \in [r-1]$ with $\langle X' \rangle = A_{i_1, i_2, \ldots, i_k}$. Let $\mathcal{F}^{X'}$ be the set of maximal $\mathcal{A}$-cuts $Y$ with respect to $\prec$ such that

- there is $R \subseteq \{i_1, i_2, \ldots, i_k\}$ with $\langle Y \rangle = A_{R(\{r\})}$ and $Y \sim_{R} X'$, and
- there are $X_i \in \mathcal{G}_i$ with $Y \sim_{ir} X_i$ for all $i \in R$.

Then update as $\mathcal{F} := \mathcal{F} \cup \{X'\} \cup \mathcal{F}^{X'}$ and $\mathcal{F}' := \mathcal{F}' \setminus \{X'\}$, and go to Step 2.

**Step 3:** Update as

$$\mathcal{F} := \text{the set of maximal elements in } \mathcal{F} \text{ with respect to } \prec.$$  

If $|\mathcal{F}| \leq 2n$, then output $\mathcal{F}$. Otherwise, output “$\mathcal{Q}$ is not compatible.”

**Proposition 2.7.** If Algorithm 3 outputs $\mathcal{F}$, then $\mathcal{F}$ displays $\mathcal{Q}$. In addition, if $\mathcal{Q}$ is compatible and $\mathcal{F}'$ is a minimal laminarizable $\mathcal{A}_{|r-1|}$-cut family displaying $\mathcal{Q}_{|r-1|}$, then $\mathcal{F}$ is a minimal laminarizable $\mathcal{A}$-cut family.

Our proposed algorithm for Displaying is the following.

Algorithm 4 (for Displaying).

**Step 1:** Execute Algorithm 2 for $\mathcal{Q}_{12}$. If Algorithm 2 returns “$\mathcal{Q}_{12}$ is not compatible,” then output “$\mathcal{Q}$ is not compatible” and stop. Otherwise, obtain $\mathcal{F}_2$.

**Step 2:** For $t = 3, \ldots, r$, execute Algorithm 3 for $\mathcal{F}_{t-1}$. If Algorithm 3 returns “$\mathcal{Q}_{t}$ is not compatible,” then output “$\mathcal{Q}$ is not compatible” and stop. Otherwise, obtain $\mathcal{F}_t$.

**Step 3:** Output $\mathcal{F} := \mathcal{F}_r$.

**Theorem 2.8.** Algorithm 4 solves Displaying in $O(rn^4)$ time. Furthermore, if the input is compatible, then the output is a minimal laminarizable $\mathcal{A}$-cut family.

3 Full multipartite quartet system

3.1 Full Displaying and Full Laminarization

As in Section 2.1, we see that Quartet Compatibility for full multipartite quartet systems can be divided into two subproblems named as Full Displaying and Full Laminarization. The outline of the argument is the same as the case of complete multipartite quartet systems in Section 2.1. We say that a family $\mathcal{F} \subseteq 2^n$ displays a full quartet system $\mathcal{Q}$ on finite set $A \subseteq [n]$ if for all distinct $a, b, c, d \in A$,

$$ab||cd \in \mathcal{Q} \iff \text{there is } X \in \mathcal{F} \text{ satisfying } a, b \in X \neq c, d \text{ or } a, b \notin X \ni c, d.$$  

Let $\mathcal{A} := \{A_1, A_2, \ldots, A_r\}$ be a partition of $[n]$ with $|A_i| \geq 2$ for all $i \in [r]$. We also say that $\mathcal{F}$ displays a full $\mathcal{A}$-partite quartet system $\mathcal{Q} = \mathcal{Q}_0 \cup \mathcal{Q}_1 \cup \cdots \cup \mathcal{Q}_r$, where $\mathcal{Q}_0$ is complete.
A-partite and \( Q_i \) is full on \( A_i \) for each \( i \in [r] \), if \( F \) displays all \( Q_0, Q_1, \ldots, Q_r \). Thus QUARTET COMPATIBILITY for full \( A \)-partite quartet system \( Q \) can also be viewed as the problem of finding a laminar family \( L \) displaying \( Q \) if it exists.

We also introduce an equivalent relation \( \approx \) on sets \( X, Y \subseteq [n] \), by: \( X \approx Y \) if the families \( \{X\} \) and \( \{Y\} \) display the same full \( A \)-partite quartet system. A set \( X \subseteq [n] \) is called a weak \( A \)-cut if \( X \neq \emptyset \). One can see that \( X \) is a weak \( A \)-cut if and only if \( X \) is an \( A \)-cut, or \( \langle X \rangle = A_i \) for some \( i \in [r] \) and \( \min(\{|X|, |A_i \setminus X|\}) \geq 2 \). One can see that, for weak \( A \)-cuts \( X, Y \), it holds that \( X \approx Y \Leftrightarrow \langle \langle X \rangle \cap X, \langle X \rangle \setminus X \rangle = \langle \langle Y \rangle \cap Y, \langle Y \rangle \setminus Y \rangle \). The equivalence relation is extended to weak \( A \)-cut families \( F, G \) by: \( F \approx G \Leftrightarrow F/\approx = G/\approx \), where \( F/\approx \) is defined as in Section 2.1. A weak \( A \)-cut family \( F \) is said to be laminarizable if there is a laminar family \( L \) with \( F \approx L \). Note that an \( A \)-cut is a weak \( A \)-cut, and for \( A \)-cuts or \( A \)-cut families, the equivalence relations \( \sim \) and \( \approx \) are the same.

By the same argument as in Section 2.1, QUARTET COMPATIBILITY for a full \( A \)-partite quartet system \( Q \) can be divided into the following two subproblems.

**Full Displaying**
Given: A full \( A \)-partite quartet system \( Q \).
Problem: Either detect the incompatibility of \( Q \), or obtain some weak \( A \)-cut family \( F \) displaying \( Q \). In addition, if \( Q \) is compatible, then \( F \) should be laminarizable.

**Full Laminarization**
Given: A weak \( A \)-cut family \( F \).
Problem: Determine whether \( F \) is laminarizable or not. If \( F \) is laminarizable, obtain a laminar weak \( A \)-cut family \( L \) with \( L \approx F \).

Here, in FULL LAMINARIZATION, we assume that no distinct \( X, Y \) with \( X \approx Y \) are contained in \( F \), i.e., \( |F| = |F/\approx| \).

FULL LAMINARIZATION can be solved in \( O(n^4) \) time by reducing to LAMINARIZATION.

**Theorem 3.1.** FULL LAMINARIZATION can be solved in \( O(n^4) \) time.

In Section 3.2, we give an \( O(rn^4) \)-time algorithm for FULL Displaying (Theorem 3.3). Thus, by Theorems 3.1 and 3.3, we obtain Theorem 1.1 for full \( A \)-partite quartet systems.

### 3.2 Algorithm for full multipartite quartet system

Our proposed algorithm for full multipartite quartet systems is devised by combining Algorithm 4 for complete multipartite quartet systems and an algorithm for full quartet systems. For full quartet system \( Q \), it is known [2] that QUARTET COMPATIBILITY can be solved in linear time of \( |Q| \), and that a phylogenetic tree displaying \( Q \) is uniquely determined. By summarizing these facts with notations introduced in this paper, we obtain the following.

**Theorem 3.2 ([2, 8]).** Suppose that \( Q \) is full on \([n]\). Then QUARTET COMPATIBILITY can be solved in \( O(|Q|) \) time. Furthermore, if \( Q \) is compatible, then a weak \([n]\)-cut family \( F \) displaying \( Q \) is uniquely determined up to \( \approx \).

Let \( A := \{A_1, A_2, \ldots, A_r\} \) be a partition of \([n]\) with \(|A_i| \geq 2\) for all \( i \in [r] \). Suppose that a full \( A \)-partite quartet system \( Q = Q_0 \cup Q_1 \cup \cdots \cup Q_r \) is compatible. Then we can obtain a minimal laminarizable \( A \)-cut family \( F_0 \) displaying \( Q_0 \) and laminar weak \( A \)-cut families \( L_i \subseteq 2^{A_i} \) displaying \( Q_i \), for \( i \in [r] \). By combining \( F_0, L_1, \ldots, L_r \) appropriately, we can construct a minimal laminarizable weak \( A \)-cut family displaying \( Q \) as follows.
Algorithm 5 (for FULL DISPLAYING).

Input: A full $A$-partite quartet system $Q = Q_0 \cup Q_1 \cup \cdots \cup Q_r$.

Output: Either detect the incompatibility of $Q$, or obtain weak $A$-cut family $F$ displaying $Q$.

Step 1: Solve DISPLAYING for $Q_0$ by Algorithm 4 and QUARTET COMPATIBILITY for $Q_i$ for $i \in [r]$. If algorithms detect the incompatibility of $Q_i$ for some $i$, then output “$Q$ is not compatible” and stop. Otherwise, obtain a $A$-cut family $F_0$ displaying $Q_0$ and laminar weak $A$-cut families $L_i \subseteq 2^{A_i}$ displaying $Q_i$ for all $i \in [r]$.

Step 2: Let $F_i := \{ X \cap A_i \mid X \in F_0 \text{ such that } \langle X \rangle \supseteq A_i \}$ for $i \in [r]$. If $F_i/\approx \not\subseteq L_i/\approx$, then output “$Q$ is not compatible” and stop.

Step 3: Define $F := F_0 \cup \bigcup_{i \in [r]} \{ Y \in L_i \mid Y \not\approx X \text{ for all } X \in F_i \}$. If $|F| \leq 2n$, then output $F$. Otherwise, output “$Q$ is not compatible.”

- Theorem 3.3. Algorithm 5 solves FULL DISPLAYING in $O(r n^4)$ time. Furthermore, if the input is compatible, then the output is a minimal laminarizable weak $A$-cut family.

By the proof of Theorem 3.3, the following corollary holds.

- Corollary 3.4. Suppose that a full $A$-partite quartet system $Q$ is compatible. Then a minimal laminarizable weak $A$-cut family $F$ displaying $Q$ is uniquely determined up to $\approx$.

References

Rectilinear Link Diameter and Radius in a Rectilinear Polygonal Domain

Elena Arseneva
St. Petersburg State University, St. Petersburg, Russia
ea.arseneva@gmail.com
https://orcid.org/0000-0002-5267-4512

Man-Kwun Chiu
Institut für Informatik, Freie Universität Berlin, Berlin, Germany
chiumk@zedat.fu-berlin.de

Matias Korman
Tufts University, Boston, USA
matias.korman@tufts.edu

Aleksandar Markovic
TU Eindhoven, Eindhoven, The Netherlands
a.markovic@tue.nl

Yoshio Okamoto
University of Electro-Communications, Tokyo, Japan
RIKEN Center for Advanced Intelligent Project, Tokyo, Japan
okamotoy@uec.ac.jp
https://orcid.org/0000-0002-9826-7074

Aurélien Ooms
Université libre de Bruxelles (ULB), Brussels, Belgium
aureooms@ulb.ac.be
https://orcid.org/0000-0002-1383

André van Renssen
University of Sydney, Sydney, Australia
andre.vanrenssen@sydney.edu.au
https://orcid.org/0000-0002-9947

Marcel Roeloffzen
TU Eindhoven, Eindhoven, The Netherlands
m.j.m.roeloffzen@tue.nl

Abstract

We study the computation of the diameter and radius under the rectilinear link distance within a rectilinear polygonal domain of $n$ vertices and $h$ holes. We introduce a graph of oriented distances to encode the distance between pairs of points of the domain. This helps us transform the problem so that we can search through the candidates more efficiently. Our algorithm computes both the

1 Partially supported by the SNF Early Postdoc Mobility grant P2TIP2-168563, Switzerland, and F.R.S.-FNRS, Belgium.
2 Supported in part by ERC StG 757609.
3 Supported in part by KAKENHI No. 17K12635, Japan and NSF award CCF-1422311.
4 Supported by the Netherlands’ Organisation for Scientific Research (NWO) under project no. 024.002.003.
5 Partially supported by JSPS KAKENHI Grant Number 15K00009 and JST CREST Grant Number JPMJCR1402, and Kayamori Foundation of Informational Science Advancement.
6 Supported by the Fund for Research Training in Industry and Agriculture (FRIA).
7 Supported by JST ERATO Grant Number JPMJER1201, Japan.
diameter and the radius in $O(\min(n^\omega, n^2 + nh \log h + \chi^2))$ time, where $\omega < 2.373$ denotes the matrix multiplication exponent and $\chi \in \Omega(n) \cap O(n^2)$ is the number of edges of the graph of oriented distances. We also provide an alternative algorithm for computing the diameter that runs in $O(n^2 \log n)$ time.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases Rectilinear link distance, polygonal domain, diameter, radius

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.58


1 Introduction

Diameters and radii are popular characteristics of metric spaces. For a compact set $S$ with a metric $d: S \times S \rightarrow \mathbb{R}^+$, its diameter is defined as $\text{diam}(S) := \max_{p \in S} \max_{q \in S} d(p, q)$, and its radius is defined as $\text{rad}(S) := \min_{p \in S} \max_{q \in S} d(p, q)$. The pair $(p, q)$ and the point $p$ that realize these distances are called the diametral pair and center, respectively. These terms are the natural extension of the same concepts in a disk and give some interesting properties of the environment, such as the worst-case response time or ideal location of a serving facility.

Much research has been devoted towards finding efficient algorithms to compute the diameter and radius for various types of sets and metrics. In computational geometry, one of the most well-studied and natural metric spaces is a polygon in the plane. This paper focuses on the computation of the diameter and the radius of a rectilinear polygon, possibly with holes (i.e., a rectilinear polygonal domain) under the rectilinear link distance. Intuitively, this metric measures the minimum number of links (segments) required in any rectilinear path connecting two points in the domain, where rectilinear indicates that we are restricted to horizontal and vertical segments only.

1.1 Previous Work

The ordinary link distance is a very natural metric and simple to describe. Initially, the interest was motivated by the potential robotics applications (i.e., having some kind of robot with wheels for which moving in a straight line is easy, but making turns is costly in time or energy). Since then, it has attracted a lot of attention from a theoretical point of view.

Indeed, many problems that are easy under the $L_1$ or Euclidean metric turn out to be more challenging under the link distance. For example, the shortest path between two points in a polygonal domain can be found in $O(n \log n)$ time for both Euclidean [9] and $L_1$ metrics [11, 12]. However, even approximating the shortest path within a factor of $(2 - \epsilon)$ under the link distance is 3-SUM hard [13], and thus it is unlikely that a significantly subquadratic-time algorithm is possible.

The problem of computing the diameter and radius is no exception to this rule: when polygons are simple (i.e., they do not have holes) and have $n$ vertices, the diameter and center can be found in linear time for both Euclidean [1, 8] and $L_1$ metrics [4]. However, the best known algorithms for the link distance run in $O(n \log n)$ time [6, 17]. Lowering the running times or proving the impossibility of this is a longstanding open problem in the field. The only partial answer to this question was given by Nilsson and Schuierer [15, 16]; they showed that the diameter and center can be found in linear time under the rectilinear link distance (i.e., when we are only allowed to use rectilinear paths).
Table 1 Summary of the best known results for computing the diameter and radius of a polygonal domain of \( n \) vertices and \( h \) holes under different metrics. In the table, \( \omega < 2.373 \) is the matrix multiplication exponent.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Simple polygon</th>
<th>Polygonal domain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Diameter</td>
<td>Radius</td>
</tr>
<tr>
<td>Euclidean</td>
<td>( O(n) ) [8]</td>
<td>( O(n) ) [1]</td>
</tr>
<tr>
<td>( L_1 )</td>
<td>( O(n) ) [4]</td>
<td>( O(n) ) [4]</td>
</tr>
<tr>
<td>Ordinary link</td>
<td>( O(n \log n) ) [17]</td>
<td>( O(n \log n) ) [6]</td>
</tr>
<tr>
<td>Rectilinear link</td>
<td>( O(n) ) [15]</td>
<td>( O(n) ) [16]</td>
</tr>
</tbody>
</table>

Figure 1 An example showing no diametral pair lies on the boundary of the polygonal domain. The points in the dashed blue regions will have distance 6 from each other (out of the 4 shortest paths connecting them two are shown) whereas other pairs will have distance 5 or less.

Figure 2 Example with diameter 8 (crossed points) and radius 7 (dotted point). By increasing the number of bends in the holes the diameter and radius become arbitrarily close. Note that any point in the domain is either a center or belongs to a diametral pair.

We focus on polygons with holes. The addition of holes to the domain introduces significant difficulties to the problem. For example, the diameter and radius under the rectilinear link distance can be uniquely realized by points in the interior of a polygonal domain (see Figure 1). Hence, it does not suffice to determine the distance only between every pair of vertices of the domain. Other strange situations can happen, such as the diameter and radius being arbitrarily close (see e.g. Figure 2).

These difficulties have a clear impact in the runtime of the algorithms. In most metrics, the runtime changes from linear or slightly superlinear to large polynomial terms. The difference between the link distance and other metrics becomes even more significant: no algorithm for computing the diameter and radius under the link distance is known, not even one that runs in exponential time (or one that works for particular cases such as rectilinear polygons). A summary of the best running time for computing the diameter and center under different metrics can be found in Table 1.

In this paper we provide the first step towards understanding such a difficult metric. Similarly to the simple polygon case [15, 16], we start by considering the computation of both the diameter and radius under the rectilinear link distance. We hope that the ideas of this paper will motivate future research in solving the more difficult problem of computing the diameter and radius under the (ordinary) link distance.
1.2 Results

Several of the difficulties of the link distance disappear when restricting the problem to a rectilinear setting. For example, one can easily partition the domain into rectangular cells such that all points in a cell have the same distance to all points in another cell. With this partition, brute-force algorithms that find the diameter and radius in \(O(n^3 \log n)\) time immediately follow. Alternatively, you can use a slightly coarser method to approximate either value: in \(O(n^2 + nh \log h)\) or \(O(n^2 \log \log n)\) time we can compute an estimate of the diameter (details of these methods are given in Section 2). This estimate will either be the exact diameter or will be the diameter plus one (i.e., the path computed may contain an additional link that is not needed).

In our work we improve this second approach. By using some geometric observations, we characterize exactly when the estimate is off by one unit. Thus, we can transform the approximation algorithm into an exact one by adding a verification step that checks whether or not the one additive error has actually happened.

We provide three different algorithms for making the above additional verification step. In Section 3 we characterize what we should look for to determine what the exact diameter is. This characterization then leads to a brute-force algorithm that runs in \(O(n^2 + nh \log h + \chi^2)\) time, where \(\chi\) is a parameter of the input that ranges from \(\Theta(n)\) to \(\Theta(n^2)\). To reduce running times when \(\chi\) is large we present another algorithm to compute the diameter in Section 4. This algorithm, which runs in \(O(n^2 \log n)\) time, exploits properties of the diameter. Specifically, we heavily use that this value is a maximum over a maximum of distances, hence it can only be used for the diameter (recall that we have a minimum-maximum alternation in the definition of the radius). For the radius we then present a third algorithm that uses matrix multiplication to speed up computation. This solution runs in time \(O(n^\omega)\), where \(\omega < 2.373\) is the matrix multiplication exponent (Le Gall [10] provided the best known bound on \(\omega\)). This last solution can also be adapted to compute the diameter, but our second algorithm results in a faster method.

Another interesting benefit of our approach is that we may be able to obtain a certificate. In previous algorithms for computing the diameter or center in polygonal domains, the diameter is found via exhaustive search. Thus, even if somehow the points that realize the diameter or center are given, the only way to verify that the answer is correct is to run the whole algorithm. In our algorithm, knowing the diameter can reduce the time needed for verification. Although the reduction in computation time is not large (from \(O(n^3 \log n)\) for computing to \(O(n^2 \log \log n)\) for verifying the diameter, for example), we find it to be of theoretical interest.

Further note that, when comparing with the algorithms for other metrics, the running time for simple and polygonal domains differs by at least a cubic factor. In our case, running times only increase by a slightly superlinear factor when compared to the case of simple polygons [15, 16]. This is partially due to the fact that rectilinear link distance is much easier than the ordinary link distance, but also because we use this new verification approach. We believe this to be our main contribution and hope that it motivates a similar approach in other metrics.

1.3 Preliminaries

A rectilinear simple polygon (also called an orthogonal polygon) is a simple polygon that has horizontal and vertical edges only. A rectilinear polygonal domain \(P\) with \(h\) pairwise disjoint holes and \(n\) vertices is a connected and compact subset of \(\mathbb{R}^2\) with \(h\) pairwise disjoint holes, in which the boundary of each hole is a simple closed rectilinear curve. Thus, the boundary \(\partial P\) of \(P\) consists of \(n\) line segments.
Each of the holes as well as the outer boundary of $P$ is regarded as an obstacle that paths in $P$ are not allowed to cross. A rectilinear path $\pi$ from $p \in P$ to $q \in P$ is a path from $p$ to $q$ that consists of vertical and horizontal segments, each contained in $P$, and such that along $\pi$ each vertical segment is followed by a horizontal one and vice versa. Recall that $P$ is a closed set, so $\pi$ can traverse the boundary of $P$ (along the outer face and any of the $h$ obstacles).

We define the link length of such a path to be the number of segments composing it. The rectilinear link distance between points $p, q \in P$ is defined as the minimum link length of a rectilinear path from $p$ to $q$ in $P$, and denoted by $\ell_{P}(p, q)$. It is well known that in rectilinear polygonal domains there always exists a rectilinear polygonal path between any two points $p, q \in P$, and thus the distance is well defined. Once the distance is defined, the definitions of rectilinear link diameter $\text{diam}(P)$ and rectilinear link radius $\text{rad}(P)$ directly follow.

For simplicity in the description, we assume that a pair of vertices do not share the same $x$- or $y$-coordinate unless they are connected by an edge. This general position assumption can be removed with classic symbolic perturbation techniques. Also notice that, since we are considering rectilinear polygons, no edge has length 0. However, for simplicity in the analysis we will allow edges in a rectilinear path to have length 0. These edges of length 0 are considered as edges and thus potentially contribute to the link distance (naturally, no shortest path will ever have such an edge). The reason for considering these is that we will consider oriented paths, where the first and last edge are forced to be horizontal or vertical, this enforcement may require edges of length 0. From now on, for ease of reading, we will refer to rectilinear simple polygons and rectilinear polygonal domains as “simple polygons” and “domains.” Similarly, we will use the term “distance” to refer to the rectilinear link distance.

## 2 Graph of Oriented Distances

In this section we introduce the graph of oriented distances and show how it can be used to encode the rectilinear link distance between points of the domain. We note that, although we have not been able to find a reference to this graph in the literature, some properties are already known. For example, the horizontal and vertical decompositions (defined below) were used by Mitchell et al. [14] to compute minimum-link rectilinear paths.

For any domain $P$, we extend any horizontal segment of the domain to the left and right until it hits another segment of $P$, partitioning it into rectangles. We call this partition the horizontal decomposition. Let $H(P)$ be the set containing those rectangles. Similarly, if we extend all the vertical segments up and down, we get the vertical decomposition. Let $V(P)$ be the set of rectangles in this second decomposition. Observe that both decompositions have linear size and can be computed in $O(n \log n)$ time with a plane sweep.

The overlay of both subdivisions creates a finer subdivision that has the well-known property that pairwise cell distance is constant (that is, the distance between any pair of points in two fixed cells of this subdivision will remain constant). Thus, by computing the distance between all pairs of cells we can find both the diameter and center. The major problem of this approach is that the finer subdivision may have $\Omega(n^2)$ cells, and thus it is hard to obtain an algorithm that runs in subcubic time. Instead, we avoid the overlay and use both subdivisions separately to obtain the diameter.

Given two rectangles $i, j \in H(P) \cup V(P)$, we use $i \cap j$ to denote the boolean operation which returns true if and only if the rectangles $i$ and $j$ properly intersect (i.e. their intersection has non-zero area). This implies that one of $i, j$ belongs to $H(P)$, and the other to $V(P)$. 

ISAAC 2018
Definition 1 (Graph of Oriented Distances). Given a rectilinear polygonal domain $P$, let $G(P)$ be the unweighted undirected graph defined as $G(P) = (\mathcal{H}(P) \cup \mathcal{V}(P), \{(h, v) \in \mathcal{H}(P) \times \mathcal{V}(P) : h \cap v\})$.

In other words, vertices of $G(P)$ correspond to rectangles of the horizontal and the vertical decompositions of $P$. We add an edge between two vertices if and only if the corresponding rectangles properly intersect. Note that this graph is bipartite, and has $O(n)$ vertices. From now on, we make a slight abuse of notation and identify a rectangle with its corresponding vertex (thus, we talk about the neighbors of a rectangle $i \in \mathcal{H}(P)$ in $G(P)$, for example).

The name Graph of Oriented Distances is explained as follows (see also the paragraph after Lemma 4). Consider a rectilinear path $\pi$ between two points in $P$. Each horizontal edge of $\pi$ is contained in a rectangle of $\mathcal{H}(P)$ and each vertical edge is contained in a rectangle of $\mathcal{V}(P)$. A bend in the path takes place in the intersection of the rectangles containing the two adjacent edges and corresponds to an edge of $G(P)$. So every rectilinear path $\pi$ has a corresponding walk $\pi'$ in $G(P)$ (and vice versa). Moreover, each bend of $\pi$ is associated with an edge of $\pi'$.

Definition 2 (Oriented distance). Given a rectilinear polygonal domain $P$, let $i$ and $j$ be two vertices of $G(P)$, let $\Delta(i, j)$ to be the length of the shortest path between $i$ and $j$ in graph $G(P)$ plus one. We also define $\Delta(i, i) = 1$.

The reason why we add the extra unit is to make sure that the link distance and the oriented distance match (see Lemma 4 below). We first list some useful properties of the oriented distance, which directly follow from the definition. Then we show the relationship between the oriented distance $\Delta(\cdot, \cdot)$ in $G(P)$ and the link distance $\ell_P(\cdot, \cdot)$ in $P$.

Lemma 3. Let $i, j, i', j'$ be any (not necessarily distinct) rectangles in $\mathcal{H}(P) \cup \mathcal{V}(P)$ such that $i \cap i'$ and $j \cap j'$. Then, the following hold.

(a) $\Delta(i, j) = \Delta(j, i)$.

(b) $\Delta(i', j') \in \{\Delta(i, j) - 1, \Delta(i, j) + 1\}$.

(c) $\Delta(i', j') \in \{\Delta(i, j) - 2, \Delta(i, j), \Delta(i, j) + 2\}$.

Lemma 4. Let $p$ and $q$ be two points of the rectilinear polygonal domain $P$. The rectilinear link distance $\ell_P(p, q)$ between $p$ and $q$ can be characterized as follows. If $p$ and $q$ lie in the same vertical or horizontal rectangle of $\mathcal{V}(P)$ or $\mathcal{H}(P)$ then $\ell_P(p, q) = 1$ (if $p$ and $q$ share a coordinate) or $\ell_P(p, q) = 2$ (if both $x$- and $y$-coordinates of $p$ and $q$ are distinct). Otherwise, let $i \in \mathcal{H}(P)$, $i' \in \mathcal{V}(P)$, $j \in \mathcal{H}(P)$ and $j' \in \mathcal{V}(P)$ be vertices of the graph of oriented distances such that $p \in i \cap i'$ and $q \in j \cap j'$. Then

$$\ell_P(p, q) = \min\{\Delta(i, j), \Delta(i', j), \Delta(i', j'), \Delta(i, j')\}.$$ 

Intuitively speaking, if we are given two disjoint rectangles $i, j \in \mathcal{H}(P)$, then $\Delta(i, j)$ denotes the minimum number of links needed to connect any two points $p \in i$ and $q \in j$ under the constraint that the first and the last segments of the path are horizontal. If we looked for rectangles in $\mathcal{V}(P)$, we would instead require that the path starts (or ends) with vertical segments. It follows that the link distance is the minimum of the four possible options.

In our algorithms we will often look for oriented distances between rectangles, so we compute it and store them in a preprocessing phase. Fortunately, a similar decomposition was used by Mitchell et al. [14]. Specifically, they show how to compute the distance from a single rectangle to all other rectangles in $O(n + h \log h)$ time with an $O(n)$-size data structure.\(^8\)

---

\(^8\) As a subproblem towards obtaining their main result, Mitchell et al. [14] show how to compute the
Lemma 5 ([14]). Given the horizontal and vertical decompositions \( H(P) \) and \( V(P) \) we can compute for a single rectangle \( i \) in either decomposition the oriented distance \( \Delta(i,j) \) to every other rectangle \( j \) in \( O(n + h \log h) \) time.

We construct this data structure for each of the \( O(n) \) rectangles. This allows us to compute (and store) the \( O(n^2) \) oriented distances in \( O(n^2 + nh \log h) \) time. Alternatively, we can use a recent result by Chan and Skrepetos [5] to compute the same distances in \( O(n^2 \log\log n) \) time.

3 Characterization via Boolean Formulas

Let \( \hat{d} = \max_{i,j \in H(P) \cup V(P)} \Delta(i,j) \) be the largest distance between vertices of \( G(P) \). Similarly, we define \( \hat{r} = \min_{i \in H(P) \cup V(P)} \max_{j \in H(P) \cup V(P)} \Delta(i,j) \). Note that these two values are the diameter and the radius of \( G(P) \) plus one (recall that we add one unit to the graph distance when defining \( \Delta \)). We use \( \hat{d} \) and \( \hat{r} \) to approximate the diameter \( \text{diam}(P) \) and radius \( \text{rad}(P) \) of a domain \( P \) under the rectilinear link distance. First, we relate the distance between two points \( p, q \in P \) to the oriented distances between the rectangles that contain \( p \) and \( q \). Specifically, from Lemma 4, we know that \( \ell(p,q) = \min\{ \Delta(i,j), \Delta(i,j'), \Delta(i',j), \Delta(i',j') \} \), where \( i,j \in H(P) \) are the horizontal rectangles containing \( p \) and \( q \), respectively, and \( i', j' \in V(P) \) are the vertical rectangles containing \( p \) and \( q \). Similarly, we define \( \hat{\ell}(p,q) = \max\{ \Delta(i,j), \Delta(i,j'), \Delta(i',j), \Delta(i',j') \} \). It then follows from Lemma 3 that these two values differ by at most 2.

Lemma 6. For any two points \( p, q \in P \), let \( i,j \in H(P) \) and \( i', j' \in V(P) \) be the rectangles containing \( p \) and \( q \), i.e., \( p \in i \cap i' \) and \( q \in j \cap j' \). Then, it holds that \( \hat{\ell}(p,q) - 2 \leq \ell(p,q) \leq \hat{\ell}(p,q) - 1 \).

This relation allows us to express the rectilinear link diameter of a domain in terms of \( \hat{d} \).

Theorem 7. The rectilinear link diameter \( \text{diam}(P) \) of a rectilinear polygonal domain \( P \) satisfies \( \text{diam}(P) = \hat{d} - 1 \) if and only if there exist \( i,i', j,j' \in H(P) \cup V(P) \) with \( i \cap i' \) and \( j \cap j' \), such that \( \Delta(i,j) = \hat{d} \) and \( \Delta(i',j') = \hat{d} \). Otherwise, \( \text{diam}(P) = \hat{d} - 2 \).

Proof. Before giving our proof, we emphasize that the fact that \( \text{diam}(P) \in \{ \hat{d} - 1, \hat{d} - 2 \} \) is folklore (although we have found no reference, several researchers mentioned that they were aware of it). Our major contribution is the characterization of which of the two cases it is.

Now observe that for any pair of points \( p,q \in P \) we have \( \ell(p,q) \leq \hat{\ell}(p,q) - 1 \leq \hat{d} - 1 \) by Lemma 6. Hence, the diameter of \( P \) is at most \( \hat{d} - 1 \). Similarly, by the definitions of \( \hat{d} \) and \( \hat{\ell}(\cdot,\cdot) \), there must be a pair of points \( p,q \in P \) so that \( \hat{\ell}(p,q) = \hat{d} \). Again by Lemma 6 it follows that \( \text{diam}(P) \geq \hat{\ell}(p,q) \geq \ell(p,q) - 2 = \hat{d} - 2 \).

Next we show that the diameter is \( \hat{d} - 1 \) if and only if the above condition holds. If \( \Delta(i,j) = \hat{d} \) and \( \Delta(i',j') = \hat{d} \), then by Lemma 3 and the fact that neither \( \Delta(i,j') \) nor \( \Delta(i',j) \) can be larger than \( \hat{d} \), we know that \( \Delta(i,j') = \Delta(i',j) = \hat{d} - 1 \). It follows from Lemma 4 that a pair of points \( p \in i \cap i' \) and \( q \in j \cap j' \) has \( \ell(p,q) = \hat{d} - 1 \). Thus, the diameter is \( \hat{d} - 1 \).
Now consider any pair $p, q$ and the set of rectangles $i, j \in \mathcal{H}(P)$ and $i', j' \in \mathcal{V}(P)$ with $p \in i \cap i'$ and $q \in j \cap j'$. Recall that $\ell_P(p, q) = \min\{\Delta(i, j), \Delta(i, j'), \Delta(j', i), \Delta(i', j')\}$. By Lemma 3, $\Delta(i, j)$ and $\Delta(i', j')$ must differ by exactly one from $\Delta(i', j)$ and $\Delta(i, j')$. That implies that two distances may be $\hat{d} - 1$, but if the condition in the lemma is not satisfied, at most one can be $\hat{d}$ and the fourth must be $\hat{d} - 2$ or less. Therefore, if the condition is not satisfied for $i, i', j, j'$, then the diameter is indeed $\hat{d} - 2$.

For the radius we can make a similar argument.

**Theorem 8.** The rectilinear link radius $\text{rad}(P)$ of a rectilinear polygonal domain $P$ satisfies $\text{rad}(P) = \hat{r} - 1$ if and only if for all $i, i' \in \mathcal{H}(P) \cup \mathcal{V}(P)$ with $i \cap i'$ there exist $j, j' \in \mathcal{H}(P) \cup \mathcal{V}(P)$ with $j \cap j'$ such that $\Delta(i, j) \geq \hat{r}$ and $\Delta(i', j') \geq \hat{r}$. Otherwise, $\text{rad}(P) = \hat{r} - 2$.

**Proof.** We first show by contradiction that the real radius satisfies $\text{rad}(P) \leq \hat{r} - 1$. Suppose the radius is greater than or equal to $\hat{r}$. Then, for all $p \in P$ there exists a point $q \in P$ such that $\ell_P(p, q) \geq \hat{r}$. Now consider a point $i \in \mathcal{H}(P) \cup \mathcal{V}(P)$, a point $p \in i$ and a point $q$ at distance $\hat{r}$ from $p$. Consider the two rectangles $j \in \mathcal{H}(P)$ and $j' \in \mathcal{V}(P)$ so that $q \in j \cap j'$. By Lemma 4 we know that $\Delta(i, j) \geq \ell_P(p, q) \geq \hat{r}$ and $\Delta(i, j') \geq \ell_P(p, q) \geq \hat{r}$. By Lemma 3b $\Delta(i, j)$ and $\Delta(i, j')$ differ by one, and thus one of them must be at least $\hat{r} + 1$. That is, for any rectangle $i$ we can find a second rectangle at oriented distance $\hat{r} + 1$. This implies that $\hat{r} = \min_{i \in \mathcal{H}(P) \cup \mathcal{V}(P)} \max_{j \in \mathcal{H}(P) \cup \mathcal{V}(P)} \Delta(i, j) \geq \hat{r} + 1$, which is a contradiction. Therefore, our initial assumption that $\text{rad}(P) \geq \hat{r}$ is false and we conclude that $\text{rad}(P) \leq \hat{r} - 1$.

Next we show that $\text{rad}(P) \geq \hat{r} - 2$. Consider any point $p$ and a rectangle $i \in \mathcal{H}(P)$ that contains it. By definition of $\hat{r}$ there is a rectangle $j \in \mathcal{H}(P) \cup \mathcal{V}(P)$ so that $\Delta(i, j) \geq \hat{r}$. Let $q$ be any point in $j$. From Lemma 6 we get that $\ell_P(p, q) \geq \hat{r}(p, q) - 2 \geq \Delta(i, j) - 2 \geq \hat{r} - 2$. Hence for any point $p$, there is a point $q$ that is at distance at least $\hat{r} - 2$, which implies $\text{rad}(P) \geq \hat{r} - 2$.

Now we show that if the above condition is satisfied, then it must hold that $\text{rad}(P) = \hat{r} - 1$. Assume the condition holds and consider any point $p$ and two rectangles $i, j \in \mathcal{H}(P) \cup \mathcal{V}(P)$ so that $i \cap i'$ and $p \in i \cap i'$. There exist $j, j' \in \mathcal{H}(P) \cup \mathcal{V}(P)$ so that $j \cap j'$, $\Delta(i, j) \geq \hat{r}$, and $\Delta(i', j') \geq \hat{r}$. By Lemma 3 we know that $\Delta(i, j')$ and $\Delta(i', j)$ must be at least $\hat{r} + 1$. Therefore $\ell_P(p, q) \geq \hat{r} - 1$ for any point $q \in j \cap j'$. This shows that for any point $p$ there is a point $q$ whose link distance to $p$ is at least $\hat{r} - 1$, giving a lower bound on the radius. Combining this with the upper bound shown above, we obtain that $\text{rad}(P) = \hat{r} - 1$ as claimed.

If the condition is not true, then we know there exist rectangles $i, i' \in \mathcal{H}(P) \cup \mathcal{V}(P)$ so that $i \cap i'$, and for every $j, j' \in \mathcal{H}(P) \cup \mathcal{V}(P)$ with $j \cap j'$ the above statement is not true. Now consider a point $p \in i \cap i'$. We argue that $p$ has distance at most $\hat{r} - 2$ to any other point $q \in P$. Consider any point $q$ and let $j, j' \in \mathcal{H}(P) \cup \mathcal{V}(P)$ be the rectangles containing $q$. We perform a case analysis on the value of $\Delta(i, j)$. First consider the case $\Delta(i, j) \geq \hat{r} + 1$. In this case $\Delta(i', j) \geq \hat{r}$ and $\Delta(i, j') \geq \hat{r}$ which contradicts our assumption that the above statement is not true for every $(j, j')$. If $\Delta(i, j) = \hat{r}$, then by Lemma 3 and the assumption that not both $\Delta(i, j) \geq \hat{r}$ and $\Delta(i', j') \geq \hat{r}$ we find that $\Delta(i, j') = \hat{r} - 2$ which implies that $\ell_P(p, q) \leq \hat{r} - 2$. If $\Delta(i, j) = \hat{r} - 1$, then by Lemma 3, both $\Delta(i, j')$ and $\Delta(i', j)$ differ from $\Delta(i, j)$ by 1, but by our assumption that not both $\Delta(i, j') \geq \hat{r}$ and $\Delta(i', j) \geq \hat{r}$, one of them must be $\hat{r} - 2$. Lastly, if $\Delta(i, j) \leq \hat{r} - 2$, we can already conclude that $\ell_P(p, q) \leq \hat{r} - 2$. This shows that from $p$ any other point $q$ is at most distance $\hat{r} - 2$ away, hence the radius is at most $\hat{r} - 2$. Combining this with the lower bound of $\hat{r} - 2$ (shown above), we conclude that the radius must be $\hat{r} - 2$.

With the above characterization, we can naively compute the diameter and the radius by checking all $O(n^4)$ quadruples $(i, i', j, j') \in \mathcal{H}(P) \times \mathcal{V}(P) \times \mathcal{H}(P) \times \mathcal{V}(P)$. However, the approach can be improved by using $G(P)$.
Corollary 9. The rectilinear link diameter $\text{diam}(P)$ and radius $\text{rad}(P)$ of a rectilinear polygonal domain $P$ consisting of $n$ vertices and $h$ holes can be computed in $O(n^2 + nh \log h + \chi^2)$ time, where $\chi$ is the number of edges of $G(P)$ (i.e., the number of pairs of intersecting rectangles of $H(P)$ and $V(P)$).

As we discuss later, this method is only useful when $\chi$ is very small, i.e. almost linear size or smaller.

Remark on the interior realization of the diameter/radius

Theorems 7 and 8 together with Lemma 3b imply that a necessary condition for the diameter to be uniquely realized by pairs of interior points is that $\text{diam}(P) = \hat{d} - 1$. Similarly, for all centers to be determined by points in the interior we must have $\text{rad}(P) = \hat{r} - 1$. However, neither condition is sufficient. This transformation of the problem into a search of quadruples of rectangles allows us to handle the interior cases in the same way as the boundary cases.

4 Computing the Diameter Faster

We present a faster method for computing the diameter. This method uses the fact that the diameter is defined as a maximum over maxima which allows us to reduce the running time to $O(n^2 \log n)$. Recall that the radius is a minimum over maxima, thus the algorithm of this section does not trivially extend to the computation of the radius. The rest of this section is the proof of the following statement.

Theorem 10. The rectilinear link diameter $\text{diam}(P)$ of a rectilinear polygonal domain $P$ of $n$ vertices can be computed in $O(n^2 \log n)$ time.

By Theorem 7, after we compute the oriented diameter $\hat{d}$, we only need to consider $\hat{d} - 1$ or $\hat{d} - 2$ as candidates to be $\text{diam}(P)$. The following corollary of Theorem 7 can be obtained by applying Lemma 3c.

Corollary 11. The diameter $\text{diam}(P)$ equals $\hat{d} - 2$ if and only if for all rectangles $i$ and $j$ with $\Delta(i, j) = \hat{d}$, and for all rectangles $i'$ and $j'$ with $i' \cap i$ and $j \cap j'$, we have $\Delta(i', j') = \hat{d} - 2$. Otherwise, $\text{diam}(P) = \hat{d} - 1$.

This condition can be checked in $O(n^4)$ time in a brute-force manner as follows. We iterate over every pair $(i, j)$ with $\Delta(i, j) = \hat{d}$. For each such pair we find the sets $\text{cover}(i) = \{i' : i \cap i'\}$ and $\text{cover}(j) = \{j' : j \cap j'\}$. Then for each pair $(i', j') \in \text{cover}(i) \times \text{cover}(j)$ we check if $\Delta(i', j') = \hat{d} - 2$. If there is a pair for which this is not the case, then by the above corollary the diameter is $\hat{d} - 1$. Since each of the covers may have linear size, the running time is $\Omega(n^4)$.

The key observation that allows us to reduce this to $O(n^2 \log n)$ time is that in the end there are only $O(n^2)$ unique pairs to test. Indeed, what we are checking is the distance of every pair $(i', j')$ in the set

$$\mathcal{T} = \{(i', j') : \exists i, j \text{ such that } (i' \cap i, j \cap j', \Delta(i, j) = \hat{d})\}$$

which clearly has only quadratic size. Next we show that this set has more structure than just being an arbitrary set of rectangles, which allows us to compute it more quickly.

First, instead of iterating over every pair $(i, j)$ with $\Delta(i, j) = \hat{d}$ and computing all pairs in $\text{cover}(i) \times \text{cover}(j)$, we iterate over $i$ and compute all pairs in $\text{cover}(i) \times \bigcup_j : \Delta(i, j) = \hat{d} \text{ cover}(j)$.
For a rectangle $i \in \mathcal{H}(P) \cup \mathcal{V}(P)$, let $\mathcal{S}_i$ denote the set of rectangles at oriented distance $\hat{d}$ from $i$. Now let

$$
\mathcal{T} = \bigcup_i \mathcal{T}_i = \bigcup_i \{(i',j'): \exists j \text{ such that } (i' \cap i, j' \cap j, j \in \mathcal{S}_i)\}.
$$

Note that the rectangles fulfilling the role of $i'$ are easily found (i.e., they must intersect $i$ and must have different orientation), but naively computing the ones that fulfill the role of $j'$ leads to a quadratic runtime. That is, if we were to compute for each $j \in \mathcal{S}_i$ its cover, then this may take $\Omega(n^2)$ time. However, there are only $O(n)$ rectangles that can fulfill the role of $j'$ and we show how to find them in $O(n \log n)$ time.

For this purpose we use an orthogonal segment intersection reporting data structure, derived from a known dynamic ray shooting data structure [7]. The data structure we use stores horizontal line segments. It allows to add or remove horizontal line segments in $O(\log n)$ time per segment. The structure reports the first segment hit by a query ray in $O(\log n)$ time. By repeatedly using the structure, we can find all $z$ horizontal line segments intersected by a vertical line segment in $O((z+1) \log n)$ time. While performing the query, we also remove all the reported segments from the data structure in the same time complexity.

For a rectangle $k$, we define the middle segment $\ell_k$ of $k$. If $k$ is a horizontal rectangle, $\ell_k$ is the line segment connecting the midpoints of its left and right boundary; if $k$ is a vertical rectangle, $\ell_k$ is the segment connecting the midpoints of its top and bottom boundary.

We fix a rectangle $i$, and assume without loss of generality that the rectangles in $\mathcal{S}_i$ are vertical. Insert the middle segments of all horizontal rectangles in $\mathcal{H}(P)$ into the intersection reporting data structure. Then, for each rectangle $j \in \mathcal{S}_i$, we query its corresponding middle segment. By the definition of middle segments, each reported horizontal segment corresponds to a rectangle $j'$ intersecting $j$. Since we remove each segment as we find it, no rectangle is reported twice. Repeating this for all $j \in \mathcal{S}_i$ finds the set $\mathcal{C}_i = \{j': j' \cap j, j \in \mathcal{S}_i\}$ of all horizontal rectangles that intersect at least one rectangle in $\mathcal{S}_i$. Each query can be charged either to the horizontal segment that is deleted from the data structure or, in case $z = 0$, to the rectangle $j \in \mathcal{S}_i$ that we are querying. Hence, the total query time sums to $O(n \log n)$.

For each rectangle in the set $\mathcal{C}_i$, we should check the distance to every rectangle $i'$ such that $i' \cap i$. Doing this explicitly takes $O(n^2)$ time. Thus, summing over all rectangles $i$, we get the total running time of $O(n^3)$.

To bring the running time down to $O(n^2 \log n)$, we create a reverse map of the map $i \mapsto \mathcal{C}_i$. For each rectangle $k$, we build a collection $\mathcal{L}_k$ that contains $i$ if and only if $k$ belongs to $\mathcal{C}_i$. Given a rectangle $j'$, we need to check the distance between $j'$ and $i'$ for any $(i,i')$ with $i \in \mathcal{L}_{j'}$ and $i' \cap i'$. Using the intersection reporting data structure, we compute for each rectangle $j'$ the set $\mathcal{D}_{j'}$, which is the set of all rectangles intersecting those in $\mathcal{L}_{j'}$. For each rectangle $i' \in \mathcal{D}_{j'}$, we test if $\Delta(i', j') = \hat{d} - 2$. Again recall that if we find a pair with $\hat{d}$, then the diameter must be $\hat{d} - 1$ (otherwise, the diameter is $d - 2$). This proves Theorem 10.

5 Computation via Matrix Multiplication

In this section we provide an alternative method to compute the radius. This method also uses the condition in Theorem 8, but instead exploits the behavior of matrix multiplication on $(0,1)$-matrices. Recall that, given two $(0,1)$-matrices $A$ and $B$, their product is $(AB)_{i,j} = \sum_k (A_{i,k} \cdot B_{k,j}) = |\{k: A_{i,k} = 1 \land B_{k,j} = 1\}|$. 


We define a $(0,1)$-matrix $I$, which is used to compute both the diameter and radius:

$$I_{i,j} = \begin{cases} 1 & \text{if } i \cap j, \\ 0 & \text{otherwise}. \end{cases}$$

In other words, for each pair $i, j$ of rectangles in $\mathcal{H}(P) \cup \mathcal{V}(P)$, the matrix $I$ indicates whether $i$ and $j$ intersect and have different orientations (one horizontal, one vertical). Note that, for ease of explanation, we have slightly abused the notation and identified rectangles of $\mathcal{H}(P) \cup \mathcal{V}(P)$ with indices in the matrix.

### 5.1 Computing the Radius

We use Theorem 8 to compute the radius. Thus, we need to determine if there exist four rectangles in $\mathcal{H}(P) \cup \mathcal{V}(P)$ that satisfy the condition of Theorem 8. If so, the radius will be $\hat{r} - 1$; otherwise, $\hat{r} - 2$. In order to do so, we define the $(0,1)$-matrix $R$ that indicates whether a pair of rectangles is at oriented distance at least $\hat{r}$ from each other:

$$R_{i,j} = \begin{cases} 1 & \text{if } \Delta(i, j) \geq \hat{r}, \\ 0 & \text{otherwise}. \end{cases}$$

By multiplying $I$ and $R$, we obtain

$$(IR)_{i,j'} = \left| \{ i' : (i \cap i') \land (\Delta(i', j') \geq \hat{r}) \} \right|.$$  

In other words, the entry at $(i, j')$ of the product $IR$ counts the number of rectangles in $\mathcal{H}(P) \cup \mathcal{V}(P)$ that intersect rectangle $i$ and are oriented differently from it, and at the same time are at oriented distance at least $\hat{r}$ from rectangle $j'$.

We construct the $(0,1)$-matrix $N$ that indicates whether the corresponding entry of $IR$ is non-zero, as follows:

$$N_{i,j} = \begin{cases} 1 & \text{if } (IR)_{i,j} > 0, \\ 0 & \text{otherwise}. \end{cases}$$

We now look at the product $RN$. Note that $(RN)_{i,j'} > 0$ if and only if there are two rectangles $j$ and $j'$ with $j \cap j'$ such that $\Delta(i, j) \geq \hat{r}$ and $\Delta(i', j') \geq \hat{r}$.

The quantifier on $j'$ and the condition on its intersection with $j$ can be moved just to the right of the quantifier on $j$ without altering the meaning of the formula, since both of them are existential quantifiers.

Therefore, the condition on Theorem 8 is satisfied if and only if for each 1-entry in $I$ the corresponding entry in $RN$ is non-zero. This condition can be checked by iterating over the entries of the matrices in quadratic time once the matrix $RN$ has been computed.

Note that the time taken by the computation of the various matrix products dominates the time taken by the other loops and operations. Each matrix has $O(n)$ rows and columns, and the product of two $O(n) \times O(n)$ matrices can be computed in $O(n^\omega)$ time. A similar method can be applied using Theorem 7 to compute the diameter instead. We summarize the results of this section in the following theorem.

▶ **Theorem 12.** The rectilinear link radius $\text{rad}(P)$ or diameter $\text{diam}(P)$ of a rectilinear polygonal domain $P$ consisting of $n$ vertices can be computed in $O(n^\omega)$ time.
References


Minimizing Distance-to-Sight in Polygonal Domains

Eunjin Oh
Max Planck Institute for Informatics Saarbrücken, Germany
eoh@mpi-inf.mpg.de

Abstract
In this paper, we consider the quickest pair-visibility problem in polygonal domains. Given two points in a polygonal domain with \( h \) holes of total complexity \( n \), we want to minimize the maximum distance that the two points travel in order to see each other in the polygonal domain. We present an \( O(n \log^2 n + h^2 \log^4 h) \)-time algorithm for this problem. We show that this running time is almost optimal unless the 3SUM problem can be solved in \( O(n^{2-\varepsilon}) \) time for some \( \varepsilon > 0 \).

2012 ACM Subject Classification
Theory of computation → Computational geometry

Keywords and phrases
Visibility in polygonal domains, shortest path in polygonal domains

1 Introduction
Consider two mobile robots under the line-of-sight communication model [8, 15]. In this model, the two robots are required to be visible to each other in order to establish communication. In the case that they are not visible to each other, we can move one of them to see the other. Motivated by this model, the quickest visibility problem was introduced. In this problem, we are given a starting point \( s \) and a target point \( t \) amidst polygonal obstacles in the plane, and the objective is to find a shortest collision-free path for \( s \) to move along to see \( t \). This problem can be solved in \( O(n \log n) \) time, where \( n \) is the total complexity of the polygonal obstacles, by applying the continuous Dijkstra paradigm [10] as mentioned in [2].

Arkin et al. [2] studied the query variant of this problem. More precisely, given \( h \) polygonal obstacles (holes) of total complexity \( n \) and a target point, they presented a data structure of size \( O(n^2 \alpha(n) \log n) \) so that the length of a shortest path for a query starting point to move along to see the target point can be computed in \( O(K \log^2 n) \) time, where \( K \) is the size of the visibility polygon from the target point and \( \alpha(n) \) is the inverse Ackermann function. Recently, it is improved by Wang [14]. His data structure has size of \( O(n \log h + h^2) \) and supports \( O(h \log h \log n) \) query time.

In this paper, we study the quickest pair-visibility problem in polygonal domains. In this problem, both starting and target points move to see each other. Precisely, given \( h \) polygonal obstacles of total complexity \( n \) and two points disjoint from the obstacles, we want to compute the minimum distance that the two points travel in order to see each other. Here, there are two variants of the problem, one for minimizing the maximum of the two travel distances and one for minimizing the sum of the two travel distances.

Wynters and Mitchell studied this problem [15] for both variants. For the min-max variant, they gave an \( O(n^3 \log n) \)-time algorithm using \( O(n^3) \) space. For the min-sum variant, they gave an \( O(nm) \)-time algorithm using \( O(m) \) space, where \( m \) is the number of edges in the visibility graph of the polygonal obstacles. Note that \( m \) is \( \Theta(n^2) \) in the worst case. Very recently, Ahn et al. [1] considered a simpler version of the quickest pair-visibility problem in which two points are given in a simple polygon with no holes, and presented linear-time
algorithms for both the min-max and the min-sum variants of the problem. They also considered a query version of the problem for the min-max variant and presented a data structure supporting $O(\log^2 n)$ query time. Both the construction time and the space of the data structure are linear in the input size.

Our results. In this paper, we study the min-max variant of the quickest pair-visibility problem in a polygonal domain with $h$ holes of total complexity $n$. We present an algorithm for this problem which takes $O(n \log^2 n + h^2 \log^4 h)$ time using $O(n \log n)$ space. This substantially improves the algorithm by Wynters and Mitchell, which takes $O(n^3 \log n)$ time using $O(n^2)$ space. Moreover, the running time of our algorithm is almost optimal unless the 3SUM problem can be solved in strongly subquadratic time. More specifically, the following lemma holds.

Lemma 1. Any algorithm for the min-max variant of the quickest pair-visibility problem in a polygonal domain with $h$ holes of total complexity $n$ takes $\Omega(n + h^{2-\varepsilon})$ time for any $\varepsilon > 0$ unless the 3SUM problem can be solved in $O(N^{2-\varepsilon})$, where $N$ is the size of input for the 3SUM problem.

Proof. We prove the lemma by introducing a reduction from a geometric version of the 3SUM problem. Given a set of $n$ points with integer coordinates on three vertical lines $x = 0$, $x = 1$ and $x = 2$, the goal of the geometric version of the 3SUM problem is to determine whether there exists a non-vertical line containing three of the points. This problem is a 3SUM-hard problem in the sense that there is an $O(n)$-time reduction from the 3SUM problem to the geometric version of the 3SUM problem [7].

Given an instance of the geometric version of the 3SUM problem, we construct a polygonal domain as follows. Let $S_i$ be the set of input points contained in the vertical line $\ell_i : x = i$ for $i = 0, 1, 2$. See Figure 1. Then $\ell_i \setminus S_i$ consists of $O(n)$ connected components (line segments or rays) in the plane. We consider each connected component as a hole of the polygonal domain. Let $g$ be the line containing the topmost point of $S_0$ and the bottommost point of $S_2$. Similarly, let $g'$ be the line containing the topmost point of $S_2$ and the bottommost point of $S_0$. We put $s$ and $t$ lying to the left of $\ell_0$ and to the right of $\ell_2$, respectively, so that $\max\{d_E(s, g), d_E(s, g'), d_E(t, g), d_E(t, g')\} \leq \min\{d_E(s, \ell_0), d_E(t, \ell_2)\}$, where $d_E(p, \ell)$ denotes the minimum Euclidean distance between a point $p$ and a point in a line $\ell$. Given $g$ and $g'$, such two points can be found in constant time.

Now consider the minimum of the maximum distance for $s$ and $t$ to travel in order to see each other. It is less than the minimum of $d(s, \ell_0)$ and $d(t, \ell_2)$ if and only if there is a non-vertical line containing three points of $S_0 \cup S_1 \cup S_2$. Therefore, if we can solve the quickest pair-visibility problem in $O(n + h^{2-\varepsilon})$ for some $\varepsilon > 0$, we can solve the geometric version of the 3SUM problem in $O(N^{2-\varepsilon})$ time, which proves the lemma. 

![Figure 1](image.png) The quickest paths for $s$ and $t$ to see each other are $ss'$ and $tt'$. Here, $s'$ and $t'$ are on a non-vertical line containing three input points.
2 Preliminaries

Consider \( h \) disjoint simple polygons in the plane of total complexity \( n \). Each polygon is considered as an open set. We let \( \mathcal{P} \) be the set of the points in the plane not contained in any of the \( h \) polygons. Here we call \( \mathcal{P} \) a polygonal domain and each polygon a hole of \( \mathcal{P} \). We say that two points \( a \) and \( b \) in \( \mathcal{P} \) are visible to each other if the line segment \( ab \) connecting \( a \) and \( b \) is contained in \( \mathcal{P} \). For a set \( A \subseteq \mathbb{R}^2 \), we use \( \partial A \) and \( \text{int}(A) \) to denote the boundary and interior of \( A \), respectively. We say a curve \( \gamma \) is convex if the Euclidean convex hull of \( \gamma \) contains \( \gamma \) on its boundary.

2.1 Geodesic Distance and Geodesic Disks

For two points \( a \) and \( b \) in \( \mathcal{P} \), there might be more than one shortest path connecting \( a \) and \( b \) in \( \mathcal{P} \). We use \( d(a, b) \) to denote the length of a shortest path between \( a \) and \( b \) contained in \( \mathcal{P} \), which we call the geodesic distance between \( a \) and \( b \). The shortest path map of a point \( x \), denoted by \( \text{SPM}(x) \), is the decomposition of \( \mathcal{P} \) into cells such that for all points \( p \) within a cell the shortest paths between \( x \) and \( p \) have the same combinatorial structure. It has complexity of \( O(n) \), and it can be constructed in \( O(n \log n) \) time using \( O(n \log n) \) space [10].

Given a point \( x \in \mathcal{P} \) and a value \( r \geq 0 \), the geodesic disk of radius \( r \) centered at \( x \), denoted by \( D_x(r) \), is defined as the set of all points in \( \mathcal{P} \) whose geodesic distances from \( x \) are at most \( r \). While \( D_x(r) \) is connected, its boundary is not necessarily connected. The boundary of \( D_x(r) \) consists of line segments and circular arcs. We call the endpoints of each maximal line segment and circular arc vertices of \( D_x(r) \). Consider the boundary of \( D_x(r) \) excluding the boundaries of all holes of \( \mathcal{P} \) and the reflex vertices of \( D_x(r) \). Each connected component is the union of circular arcs of \( D_x(r) \). We call each connected component a geodesic spiral of \( D_x(r) \). Given \( \text{SPM}(x) \), we can construct \( D_x(r) \) for a fixed \( r \geq 0 \) in \( O(n) \) time by considering all cells of \( \text{SPM}(x) \) one by one.

2.2 Extended Corridor Structure

Our algorithm uses the extended corridor structure of a domain [3, 4, 11]. A hole in the domain we will consider in this paper is either a simple polygon (a hole of \( \mathcal{P} \)) or a splinegon which is a part of \( D_s(r) \) and \( D_t(r) \) for two input points \( s \) and \( t \) in \( \mathcal{P} \) and a fixed value \( r \geq 0 \). Each hole is considered as an open set. A splinegon is defined as a set obtained from a simple polygon \( P \) by replacing each edge \( e \) of \( P \) with a curved edge \( e' \) joining the endpoints of \( e \) such that the region bounded by \( e \) and \( e' \) is convex [6]. Thus a simple polygon itself is also a splinegon. A splinegon is said to be simple if an edge intersects another edge only at their common endpoint. A domain having splinegons as its holes is called a splinegon domain.

Chen and Wang [3] studied a decomposition of a splinegon domain \( Q \) which is called the extended corridor structure. They first considered a bounded degree decomposition of \( Q \), which is a subdivision of \( Q \) into cells each with at most four sides and with at most three neighboring cells. They presented an \( O(n + h \log^{1+\varepsilon} h) \)-time algorithm for computing a bounded degree decomposition of \( Q \) into \( O(n) \) cells for any \( \varepsilon > 0 \), where \( h \) is the number of the splinegons in the domain and \( n \) is the total complexity of the splinegons. Such a subdivision is achieved by adding \( O(n) \) nonintersecting diagonals. See Figure 2(a).

In our case, the boundary of a hole might overlap with the boundary of another hole while the holes are pairwise interior-disjoint. The algorithm by Chen and Wang still works for this case. In this case, an edge of the common boundary of two holes is considered as a (degenerate) cell. In this way, \( Q \) coincides with the union of the closures of the cells of the
Minimizing Distance-to-Sight in Polygonal Domains

Figure 2 (a) A bounded degree decomposition. The gray regions are junction regions with non-empty interiors. (b) An hourglass and four bays. (c) Two funnels, two canals and two bays.

bounded degree decomposition. The dual graph of the bounded degree decomposition is a planar graph such that the degree of each node is at most three. The cell corresponding to a node of degree 3 in the dual graph is called a junction region. It is known that the number of the junction regions in \( Q \) is \( O(h) \) \[3, 11\].

Imagine that we remove the closures of all junction regions from \( Q \), which partitions \( Q \) into a number of connected regions. Each connected region is called a corridor. If a corridor has an empty interior, it lies on the common boundary of two holes. A corridor \( C \) with non-empty interior is a simple splinegon and has two boundary edges, say \( ab \) and \( cd \), each incident to a junction region. We call them the gates of \( C \). The boundary of \( C \) other than the gates consists of two chains connecting the gates such that each chain is a part of the boundary of a hole incident to \( C \). See Figure 2(b–c).

Since a corridor is a simple splinegon, the shortest path connecting two points in \( C \) is unique. Let \( \pi_C(x, y) \) denote the shortest path connecting two points \( x \) and \( y \) in \( C \). For the gates \( ab \) and \( cd \) such that \( a, b, c, d \) appear on the boundary of \( C \) in the order, let \( H_C \) be the region bounded by \( ab, cd \) and \( \pi_C(a, d) \) and \( \pi_C(b, c) \). If \( \pi_C(a, d) \) and \( \pi_C(b, c) \) are disjoint, \( H_C \) is called an hourglass of \( C \). See Figure 2(b). Otherwise, the interior of \( H_C \) consists of two connected components, each of which is called a funnel of \( C \). See Figure 2(c). For both cases, we call a connected component \( R \) of \( C \setminus H_C \) a bay if it is incident to exactly one edge of \( \pi_C(a, d) \cup \pi_C(b, c) \). Otherwise, we call it a canal. We call an edge of \( \pi_C(a, d) \cup \pi_C(b, c) \) incident to \( R \) (a bay or a canal) a gate of \( R \). Also, we call \( \pi_C(a, d) \) and \( \pi_C(b, c) \) the corridor paths.

The union of the closures of the junction regions, hourglasses and funnels is called the ocean of \( Q \). It consists of \( O(h) \) convex chains with a total of \( O(n) \) vertices. Notice that the ocean is not necessarily connected. In this way, the interior of \( Q \) is subdivided into the ocean, bays and canals. We call this subdivision the extended corridor structure of \( Q \). Given a bounded degree decomposition of \( Q \), one can compute the extended corridor structure in \( O(n) \) time \[3\]. This structure has been used as a tool for various types of visibility problems due to the following property.

\begin{itemize}
  \item Lemma 2 ([4, The Opaque Property]). \textbf{For any canal, suppose a line segment \( pq \) is in \( Q \) such that neither \( p \) nor \( q \) is in the canal. Then \( pq \) does not contain any point of the canal that is not on its two gates.}
\end{itemize}

Since the part of a corridor path incident to a canal is convex, we have the following lemma.

\begin{itemize}
  \item Lemma 3. \textbf{For a canal, consider a line segment \( pq \subset Q \) intersecting a gate of the canal. Then \( pq \) intersects the part of a corridor path incident to the canal only at points on its gates.}
\end{itemize}
2.3 Sketch of the Algorithm

In this paper, we consider the min-max variant of the quickest pair-visibility problem. Given a polygonal domain $\mathcal{P}$ with $h$ holes of complexity $n$ and two points $s$ and $t$ in $\mathcal{P}$, the objective is to compute two paths in $\mathcal{P}$, one for $s$ and one for $t$, to travel in order to see each other such that the maximum of the path lengths is minimized. Let $r^*$ be the optimal solution, that is, the minimum of $\max\{d(s, s'), d(t, t')\}$ among all pairs $(s', t')$ such that $ss'$ and $tt'$ are visible to each other. Then to obtain $r^*$, we apply the parametric search technique by using the decision algorithm as a subroutine.

For the decision problem, we assume that we have $\text{spm}(s)$ and $\text{spm}(t)$. Notice that they are independent of an input distance $r$. Also, we further assume that $r$ is less than $d(s, t)/2$, that is, $D_s(r)$ and $D_t(r)$ are disjoint. Note that there is a point at distance $d(s, t)/2$ from each of $s$ and $t$ if $r \geq d(s, t)/2$. In this case, $s$ and $t$ can see each other by moving to this point. Therefore the answer is positive for any $r \geq d(s, t)/2$.

3 Decision Problem

For a fixed $r > 0$, the decision problem asks if there are two points $s'$ and $t'$ in $\mathcal{P}$ such that $d(s, s') \leq r$, $d(t, t') \leq r$, and $s'$ and $t'$ are visible to each other. If so, we say $r$ is feasible. Such a segment $s't'$ always intersects geodesic spirals of $D_s(r)$ and $D_t(r)$. Thus there always exists a segment $s't' \subseteq \mathcal{P}$ intersecting $D_s(r)$ only at $s'$ and intersecting $D_t(r)$ only at $t'$ if $r$ is feasible. We call such a segment for a feasible value $r$ a witness segment for $r$. Notice that one endpoint of a witness segment lies on a geodesic spiral of $D_s$, and the other endpoint lies on a geodesic spiral of $D_t$. In this section, we present an $O(n + h^2 \log^2 h)$-time algorithm for deciding if a given value $r$ is feasible. Since $r$ is fixed, we simply let $D_s = D_s(r)$ and $D_t = D_t(r)$. Recall that $D_s$ and $D_t$ are disjoint by the assumption that $r < d(s, t)/2$.

3.1 Finding $O(h)$ Geodesic Spirals from Each Geodesic Disk

We want to construct the extended corridor structure of $\mathcal{P}\backslash(\text{int}(D_s) \cup \text{int}(D_t))$, but $D_s$ and $D_t$ are not necessarily splinegons because their boundaries are not necessarily connected. Moreover, it is possible that they have $\Theta(n)$ geodesic spirals even if $h = 1$. To decide if $r$ is feasible efficiently, we choose a subset $\bar{D}_s$ (and $\bar{D}_t$) of $D_s$ (and $D_t$) which is a splinegon containing $O(h)$ geodesic spirals of $D_s$ (and $D_t$) on its boundary, and consider the extended corridor structure of $\mathcal{P}\backslash(\text{int}(\bar{D}_s) \cup \text{int}(\bar{D}_t))$.

Consider $\mathcal{P}\backslash D_s$, which consists of $O(n)$ connected subregions. Since $D_s$ and $D_t$ are disjoint, $D_t$ is contained in exactly one of such subregions. Moreover, no witness segment intersects subregions of $\mathcal{P}\backslash D_s$ other than the one containing $D_t$. See Figure 3. Therefore, it suffices to consider the subregion containing $D_t$ only. We can find the subregion containing $D_t$ in $O(n)$ time as follows. Using $\text{spm}(s)$, we compute a shortest path connecting $s$ and $t$, and find the point in the path whose distance from $s$ is $r$. This point is contained in the boundary of the subregion. Starting from this point, we walk along the boundary of the subregion until we reach this point again in time linear in its complexity, which is $O(n)$.

Consider the common boundary of $D_s$ and the subregion of $\mathcal{P}\backslash D_s$ containing $D_t$. It is contained in a connected component, say $\eta$, of the boundary of $D_s$. Moreover, it consists of geodesic spirals of $D_s$, appearing on $\eta$ consecutively. Let $a$ and $b$ be the most clockwise and counterclockwise points on such geodesic spirals. Let $\bar{D}_s$ denote the region bounded by
a shortest path between $s$ and $a$, a shortest path between $s$ and $b$, and the part of $\eta$ lying between $a$ and $b$ and containing the geodesic spirals on the common boundary. See Figure 3. We choose an arbitrary shortest path between $s$ and $a$ (or $b$) if it is not unique. Here, $\overline{D}_s$ might contain a hole of $P$. In this case, we ignore such holes. In the following, we assume that no hole of $P$ intersects the interior of $\overline{D}_s$. We define $\overline{D}_t$ in the same way by changing the roles of $s$ and $t$. Then a witness segment intersects $\overline{D}_s$ only at a point on $\overline{D}_s$ and intersects $\overline{D}_t$ only at a point on $\overline{D}_t$. Also, we have the following lemma.

Lemma 4. $\overline{D}_s$ and $\overline{D}_t$ contain $O(h)$ geodesic spirals of $D_s$ and $D_t$, respectively, on their boundaries.

Proof. We prove the lemma for $\overline{D}_s$ only. The case of $\overline{D}_t$ can be proved analogously.

An endpoint of a geodesic spiral of $\overline{D}_s$ is a point on the boundary of a hole of $P$ or a reflex vertex of $D_s$. We claim that for each hole $H$ of $P$, at most two geodesic spirals of $\overline{D}_s$ have their endpoints on the boundary of $H$. We also claim that there are $O(h)$ reflex vertices of $D_s$ lying on the boundary of $\overline{D}_s$. These two claims imply the lemma.

For the first claim, consider a hole $H$ of $P$. Assume to the contrary that there are three geodesic spirals, say $\gamma_1, \gamma_2$ and $\gamma_3$, having their endpoints on the boundary of $H$. Recall that the geodesic spirals of $\overline{D}_s$ are incident to the same connected component of $P \setminus \overline{D}_s$, say $R$. Therefore, the boundary of $H$ appears on the boundary of $R$ at least twice. Notice that $H$ is a simple polygon, which is connected. This means that $\overline{D}_s$ is disconnected, which is a contradiction.

For the second claim, let $v$ be a reflex vertex of $\overline{D}_s$ lying on the boundary of $\overline{D}_s$. Let $\beta_1$ and $\beta_2$ be the circular arcs of $\overline{D}_s$ incident to $v$. Consider a shortest path $\pi_i$ connecting the center of $\beta_i$ and $s$ for $i = 1, 2$. If there are more than one shortest path, we choose the one so that the region bounded by $\pi_1, \pi_2$, and the two line segments connecting $v$ and the centers of $\beta_i$ is minimized. Such a region contains a hole of $P$ by construction. Moreover, such regions for all reflex vertices of $\overline{D}_s$ lying on the boundary of $\overline{D}_s$ are pairwise interior disjoint by the choice of $\pi_i$. Since each such region contains a hole of $P$, there are at most $h$ reflex vertices of $\overline{D}_s$ lying on the boundary of $\overline{D}_s$. Therefore, the lemma holds.

3.2 Extended Corridor Structure of the Splinegon Domain

We construct the extended corridor structure of $Q = P \setminus (\text{int}(\overline{D}_s) \cup \text{int}(\overline{D}_t))$ in $O(n + h \log^{1+\varepsilon} h)$ time for any $\varepsilon > 0$ [3]. Notice that we consider the interiors of $\overline{D}_s$ and $\overline{D}_t$ as holes of $Q$. The boundary of the ocean of $Q$ consists of $O(h)$ convex curves each of which consists of a part of a single hole of $Q$ or a part of a corridor path.

Recall that an (straight or circular) arc of the ocean is a part of the boundary of the holes of $Q$ or a gate of a bay or a canal. Consider the arcs of the ocean which are gates of the bays and canals defined by $\overline{D}_s$ and $\overline{D}_t$. By construction, the boundary of each such
bay or canal consists of its gates and geodesic spirals of $\bar{D}_s$ or $\bar{D}_t$ only. Therefore, there are $O(h)$ such arcs of the ocean by Lemma 4 and the fact that a geodesic spiral contains a reflex vertex of $\bar{D}_s$ or $\bar{D}_t$ only at its endpoints. Imagine that we remove the gates of the bays and canals defined by $\bar{D}_s$ and $\bar{D}_t$ from the boundary of the ocean. The remaining part of the boundary still consists of $O(h)$ convex curves. Let $\Gamma$ be the set of such convex curves and the gates of the bays and canals defined by $\bar{D}_s$ and $\bar{D}_t$. Note that the union of all curves and gates in $\Gamma$ is the boundary of the ocean. A curve of $\Gamma$ is defined by $\bar{D}_s$ (or $\bar{D}_t$) if it lies on the boundary of $\bar{D}_s$ (or $\bar{D}_t$) or it is a gate of a bay or a canal defined by $\bar{D}_s$ (or $\bar{D}_t$).

The following lemmas are keys of our decision algorithm. Due to them, it suffices to consider the curves of $\Gamma$ only.

**Lemma 5.** If a witness segment does not intersect the closure of the ocean, it is contained in the interior of a corridor defined by $\bar{D}_s$ and $\bar{D}_t$. Moreover, if such a corridor exists, $r$ is feasible.

**Proof.** If a witness segment $\ell$ does not intersect the closure of the ocean, it is contained in a corridor, say $C$. By construction, the boundary of $C$ consists of parts of the boundaries of two holes and two gates. Since the endpoints of $\ell$ are on geodesic spirals of $\bar{D}_s$ and $\bar{D}_t$, the holes defining $C$ are $\bar{D}_s$ and $\bar{D}_t$. Now assume that a corridor defined by $\bar{D}_s$ and $\bar{D}_t$ exists. Then a gate of the corridor connects a point of a geodesic spiral of $\bar{D}_s$ and a point of a geodesic spiral of $\bar{D}_t$. In other words, such a gate is a witness segment, and thus $r$ is feasible.

**Lemma 6.** If a witness segment $\ell$ intersects the closure of the ocean, the intersection between $\ell$ and the ocean is a line segment whose endpoints are on curves of $\Gamma$ defined by $\bar{D}_s$ and $\bar{D}_t$.

**Proof.** Let $\ell'$ be the intersection between $\ell$ and the ocean. By construction, a connected component of $\ell \setminus \ell'$ is contained in a bay or a canal by the opaque property. Thus $\ell \setminus \ell'$ consists of at most two connected components, and $\ell'$ is a line segment. Let $p$ be an endpoint of $\ell'$. If $p$ is an endpoint of $\ell$, it lies on a convex curve of $\Gamma$ contained on a geodesic spiral of $\bar{D}_s$ or $\bar{D}_t$, and the lemma holds. Thus we assume that $p$ is not an endpoint of $\ell$. Then the connected component of $\ell \setminus \ell'$ incident to $p$ is contained in a bay or a canal defined by $\bar{D}_s$ or $\bar{D}_t$. This means that $p$ lies on a gate of the bay or canal, and therefore, it lies on the curve of $\Gamma$ defined by $\bar{D}_s$ or $\bar{D}_t$.

**Lemma 7.** If $r$ is feasible and no corridor is defined by $\bar{D}_s$ and $\bar{D}_t$, there is a witness segment $\ell$ such that the intersection between $\ell$ and the ocean is tangent to a curve of $\Gamma$ or connects an endpoint of a curve of $\Gamma$ defined by $\bar{D}_s$ and an endpoint of a curve of $\Gamma$ defined by $\bar{D}_t$.

**Proof.** Let $\ell'$ be the intersection between $\ell$ and the ocean. Its endpoints are contained in curves $\gamma_s$ and $\gamma_t$ of $\Gamma$ defined by $\bar{D}_s$ and $\bar{D}_t$, respectively, by Lemma 6. We move one endpoint of $\ell'$ in clockwise direction along the curve $\gamma_s$ of $\Gamma$ containing it until (1) $\ell'$ (excluding its endpoints) contains a vertex of the ocean, (2) $\ell'$ intersects $\gamma_s$ (or $\gamma_t$) at a point other than the endpoints of $\ell'$, or (3) the endpoint reaches an endpoint of $\gamma_s$. For Case (1), $\ell'$ is tangent to a curve of $\Gamma$, and thus we are done. For Case (2), $\ell'$ is tangent to $\gamma_s$ (or $\gamma_t$), and thus we are done. For Case (3), we move the other endpoint of $\ell'$ in the same way. Then the lemma holds.

We call the intersection between a witness segment satisfying Lemma 7 and the ocean an ocean-restricted witness segment.
3.3 Finding a Witness Segment: Rotating Lines around Convex Curves

We assume that no corridor is defined by $\bar{\mathcal{D}}_s$ and $\bar{\mathcal{D}}_t$. Otherwise, we return the positive answer immediately by Lemma 5. Our goal in this subsection is to find an ocean-restricted witness segment if $r$ is feasible. By definition (and by Lemma 7), an ocean-restricted witness segment is tangent to a convex curve of $\Gamma$ or contains an endpoint of a curve of $\Gamma$. We check for each convex curve $\gamma$ of $\Gamma$ if there is an ocean-restricted witness segment tangent to $\gamma$. In a similar way, we check for each endpoint of the curves of $\Gamma$ if it contains an ocean-restricted witness segment.

Imagine that we rotate a line tangent to $\gamma$ along $\gamma$. More specifically, let $\langle e_1, \ldots, e_k \rangle$ be the sequence of the (circular or straight) arcs of $\gamma$ in order. For an integer $i$, let $v_i$ and $v_{i+1}$ be the endpoints of $e_i$. The process will be initialized with the line tangent to $e_1$ at $v_1$. It rotates along $e_1$ until it hits $v_2$. Then it rotates around $v_2$ (while remaining tangent to $\gamma$ at $v_2$) until it is tangent to $e_2$. In general, the current line is rotated around $v_i$ in a way so that it remains tangent to $\gamma$ at $v_i$ until it is tangent to $e_i$, and then it rotates along $e_i$ until it hits $v_{i+1}$. The process is iterated with $v_{i+1}$ as the new rotation center. The process terminates as soon as the line is tangent to $\gamma$ at $v_{k+1}$. If an ocean-restricted witness segment is tangent to $\gamma$, we encounter the line containing it during the process.

In the following, for each line $\ell$ we encounter during the process, we let $\ell^+$ and $\ell^-$ be the connected components (rays) of $\ell \setminus \gamma$ such that $\ell^+$ goes towards $v_{k+1}$ and $\ell^-$ goes towards $v_1$. See Figure 4. An ocean-restricted witness segment is contained in $\ell$ if and only if the first curve of $\Gamma$ hit by $\ell^+$ is defined by one of $\mathcal{D}_s$ and $\mathcal{D}_t$, and the first curve of $\Gamma$ hit by $\ell^-$ is defined by the other geodesic disk. We show how to maintain the first curve of $\Gamma$ hit by $\ell^+$ only. We can do this for $\ell^-$ analogously.

More generally, we maintain the sequence $S$ of the curves of $\Gamma$ hit by $\ell^+$ in order. Since every curve of $\Gamma$ is convex, it appears on $S$ at most twice. During the sweep, for each convex curve $\gamma'$ of $\Gamma$, there are at most four events where the number of appearances of $\gamma'$ on $S$ changes. See Figure 4. Moreover, such events (rays) are on common tangents of $\gamma$ and $\gamma'$ or lines tangent to $\gamma$ which pass through an endpoint of $\gamma'$. We can compute the common tangents of $\gamma$ and $\gamma'$ $O(\log h)$ time if the arcs of each convex curve are stored in a balanced binary search tree [13]. Similarly, we can compute the lines tangent to $\gamma$ and passing through a specific point in $O(\log h)$ time [13]. We can construct the balanced binary search trees of the curves of $\Gamma$ in time linear in their complexities after computing $\Gamma$. Since no convex curve of $\Gamma$ crosses another convex curve of $\Gamma$, the sequence $S$ changes only at these events. Thus there are $O(h)$ events in total, and we can obtain and sort all events in $O(h \log h)$ time. We can handle each event in $O(\log^2 n)$ time as follows.

![Figure 4](image-url) Any ray tangent to $\gamma$ lying between $\ell^+_1$ and $\ell^+_2$ intersects $\gamma'$ twice and any ray tangent to $\gamma$ lying between $\gamma'_2$ and $\gamma'_3$ intersects $\gamma'$ once.
When we encounter a new event $\ell^+$, the convex curve $\gamma'$ defining $\ell^+$ disappears from $S$ or appears on $S$. For the case that it disappears from $S$, we update $S$ accordingly in $O(\log n)$ time. For the other case, we apply binary search on the elements of $S$ to find the position of the new appearance of $\gamma$ on $S$. In each iteration of the binary search, we want to compute the order of the points (at most four points) in $\ell^+ \cap \gamma'$ and $\ell^+ \cap \gamma''$ along $\ell^+$ for some curve $\gamma''$ appearing on the current sequence $S$. We can compute the points in $O(\log n)$ time by a straightforward binary search on the arcs of $\gamma'$ (and $\gamma''$), and then sort them in $O(1)$ time. This gives the position of the new appearance of $\gamma'$ with respect to $\gamma''$. After $O(\log n)$ iterations, we can find the position of the new appearance of $\gamma'$ on $S$. Then we update $S$ accordingly. In this way, we can handle each event in $O(\log^2 n)$ time.

After rotating a line along $\gamma$, we can check if an ocean-restricted witness segment is contained in a line we encountered so far. Since we have $O(h)$ curves of $\Gamma$, the total time for checking if an ocean-restricted witness segment is tangent to a curve of $\Gamma$ is $O(h^2 \log^2 h)$.

Also, we check for each endpoint of the curves of $\Gamma$ if it contains an ocean-restricted witness segment. We can do this in a similar way: rotate a line around this endpoint. Therefore, we have the following lemma.

**Lemma 8.** Given a value $r > 0$, we can check if there are two points $s'$ and $t'$ such that $d(s, s') \leq r$, $d(t, t') \leq r$, and $s'$ and $t'$ are visible to each other in $O(n + h^2 \log^2 h)$ time assuming that SPM$(s)$ and SPM$(t)$ are given.

### 4 Optimization Problem

Let $(s^*, t^*)$ be a pair of points in $P$ that minimizes the maximum of $d(s, s^*)$ and $d(t, t^*)$ such that $s^*$ and $t^*$ are visible to each other. Let $r^*$ be the maximum of $d(s, s^*)$ and $d(t, t^*)$. In this section, we compute $(s^*, t^*)$ and $r^*$ by applying parametric search technique [12].

Basically, we apply the decision algorithm with input $r^*$ without explicitly computing $r^*$. In the decision algorithm, we maintain a number of structures including geodesic disks, the splinegon domain $Q$ and the sequence $\Gamma$ which depend on an input distance $r$. In this section, we consider such structures as functions of $r$. For example, we use $\Gamma(r)$ to denote the sequence $\Gamma$ for an input distance $r$. While the algorithm described in this section is executed, we maintain an interval $[\gamma_1, \gamma_2]$ containing $r^*$ so that the combinatorial structures of structures we have computed so far remain the same for every $r \in [\gamma_1, \gamma_2]$. Then we will see that $\gamma_1$ becomes $r^*$ for the interval we have at the end.

#### 4.1 Combinatorial Structures of $\bar{D}_s$ and $\bar{D}_t$

The first step of the decision algorithm is to compute $D_s(r)$, $D_t(r)$, $\bar{D}_s(r)$ and $\bar{D}_t(r)$. Here, we compute their combinatorial structures for $r = r^*$ instead of computing them explicitly.

We first compute the combinatorial structures of $D_s(r^*)$ and $D_t(r^*)$. Notice that the endpoints of the circular arcs of $D_s(r^*)$ and $D_t(r^*)$ lie on edges of SPM$(s)$ and SPM$(t)$, respectively. Also, the boundary of $D_s(r^*)$ is not necessarily connected. For a radius $r > 0$, the combinatorial structure of $D_s(r)$ is defined as a set of the sequences of edges of SPM$(s)$ such that each sequence consists of the edges of SPM$(s)$ intersecting a connected component of the boundary of $D_s(r)$ in the clockwise order along the component.

For each vertex of SPM$(s)$, we compute the geodesic distance between the vertex and $s$. Also for each edge of $P$, we compute the smallest geodesic distance between a point on the edge and $s$. We can compute them in $O(n)$ time by considering all cells of SPM$(s)$ one by one. Then we sort all distances in increasing order in $O(n \log n)$ time. We find
the smallest interval \([r_1, r_2]\) containing \(r^*\) for two distances \(r_1\) and \(r_2\) we obtained by applying binary search on all such distances with the decision algorithm. Since the decision algorithm takes \(O(n + h^2 \log^2 h)\) time assuming that we have \(\text{SPM}(s)\) and \(\text{SPM}(t)\), we can find \([r_1, r_2]\) in \(O(n \log n + h^2 \log^2 h \log n) = O(n \log n + h^2 \log^3 h)\) time in total. For any radius \(r \in [r_1, r_2]\), the combinatorial structure of \(D_s(r)\) remains the same. We also compute the combinatorial structure of \(D_t(r)\), and update \([r_1, r_2]\) containing \(r^*\) so that for any \(r \in [r_1, r_2]\), the combinatorial structure of \(D_t(r)\) (and \(D_s(r)\)) remains the same.

Also, we define the combinatorial structure of \(D_s(r)\) to be the sequence of the edges of \(\text{SPM}(s)\) intersecting the boundary of \(D_s(r)\) in clockwise order. For any \(r \in [r_1, r_2]\), the combinatorial structure of \(D_s(r)\) remains the same by the definition of \(D_s(r)\). The same holds for \(D_t(r)\).

By construction, an endpoint of a circular arc of \(D_s(r)\) is represented as an algebraic function of constant complexity for a value \(r \in [r_1, r_2]\). We obtain the splinegon domain \(Q(r)\) defined by \(\text{int}(D_s(r))\), \(\text{int}(D_t(r))\) and the holes of \(P\) for \(r \in [r_1, r_2]\). Here, a vertex of \(Q\) is represented as an algebraic function with respect to \(r\).

### 4.2 Combinatorial Structure of the Extended Corridor Structure

To construct the extended corridor structure of a splinegon domain, the algorithm by Chen and Wang [3] computes a bounded degree decomposition of the splinegon domain. Then based on this, they compute the extended corridor structure. In the following, we split each arc of \(Q(r)\) into at most four pieces so that it is monotone with respect to the \(x\)-axis and \(y\)-axis. In other words, we add at most three vertices to each arc. Here each of the new vertices is also represented as an algebraic function with respect to \(r\).

**Bounded degree decomposition.** The algorithm by Chen and Wang [3] first decomposes the domain with respect to the horizontal extensions obtained from each hole vertex of \(P\) going in both directions until they hit the boundary of the domain. Then each cell has at most four sides, but it might have more than three neighboring cells. In such a case, the algorithm splits each such cell further with respect to vertical extensions from vertices of the cell. Then it splits each of the resulting cells further with respect to its diagonal if it still has more than three neighboring cells. In our case, we want to represent the vertices of the bounded degree decomposition of \(Q(r)\) as algebraic functions with respect to \(r\) for \(r \in [r_1', r_2']\) for some interval \([r_1', r_2'] \subseteq [r_1, r_2]\) containing \(r^*\).

Suppose that the order of the vertices of \(Q(r)\) with respect to the \(y\)-axis is the same for any \(r \in [r_1', r_2']\) for some interval \([r_1', r_2'] \subseteq [r_1, r_2]\) containing \(r^*\). Moreover, the order of the vertices of \(Q(r)\) with respect to the \(x\)-axis is the same for any \(r \in [r_1', r_2']\). Then the combinatorial structure of the bounded degree decomposition of \(Q(r)\) remains the same for any \(r \in [r_1', r_2']\). In other words, a vertex of the bounded degree decomposition of \(Q(r)\) is represented as an algebraic function of \(r\) for \(r \in [r_1', r_2']\). Thus in the following, we show how to sort the vertices of \(Q(r)\) with respect to the \(x\)-axis.

To do this, we use Cole’s sorting algorithm which sorts \(m\) elements in \(O(\log m)\) iterations each consisting of \(O(m)\) comparisons [5]. Here, the comparisons in each iteration is independent to one another. In our case, we are to sort all vertices of \(Q(r)\) with respect to the \(x\)-axis. Here, \(m = O(n)\). For each iteration, we complete \(O(n)\) comparisons of vertices of \(Q(r)\) as follows. Suppose that we are to compare two vertices \(v_1(r)\) and \(v_2(r)\) represented by algebraic functions of \(r \in [r_1, r_2]\). The result of the comparison changes only at \(O(1)\) times as \(r\) changes from \(r_1\) to \(r_2\). We obtain \(O(1)\) such values in \(O(1)\) time. We do this for all of the \(O(n)\) comparisons, and then we have \(O(n)\) values. Then we apply binary
search on the values so that we find an interval \([r_1', r_2'] \subseteq [r_1, r_2]\) containing \(r^*\) and the result of each comparison remains the same for any \(r \in [r_1', r_2']\). We can find the interval in \(O(T_p \log n) = O(n \log n + h^2 \log^3 h)\) time, where \(T_p\) is the running time for the decision algorithm. We update the interval \([r_1, r_2]\) that we maintain to \([r_1', r_2']\). After completing \(O(\log n)\) iterations, we obtain the interval \([r_1, r_2]\) containing \(r^*\) such that the sorted list of the vertices of \(Q(r)\) remains the same for every \(r \in [r_1, r_2]\). Thus we can sort all vertices with respect to the \(x\)-axis (and the \(y\)-axis) in \(O(n \log^2 n + h^2 \log^4 h)\) time in total, and we can obtain the combinatorial structure of the bounded degree decomposition of \(Q(r^*)\) in the same time.

**Extended corridor structure.** The next step for computing the extended corridor structure is to compute the shortest paths for each corridor. We can make this procedure parallelized using the algorithm \([9]\). More precisely, this algorithm computes the shortest path between two points in \(O(\log N)\) iterations each consisting of \(O(N)\) comparisons. As we did for the bounded degree decomposition, we can reduce the interval \([r_1, r_2]\) containing \(r^*\) so that the combinatorial structure of the extended corridor structure remains the same for any \(r \in [r_1, r_2]\). Since this can be done in \(O(\log n)\) iterations each consisting of \(O(n)\) steps for all corridors of \(Q(r^*)\), we can compute the combinatorial structure of the extended corridor structure in \(O(n \log^2 n + h^2 \log^4 h)\) time as we did for computing the bounded degree decomposition.

- **Lemma 9.** We can obtain the combinatorial structure of the extended corridor structure of \(Q(r^*)\) in \(O(n \log^2 n + h^2 \log^4 h)\) time.

Since we have the combinatorial structure of \(Q(r^*)\), we can obtain \(\Gamma(r^*)\) in the same time. Again, an endpoint of each convex curve of \(\Gamma(r)\) is represented as an algebraic function of \(r\).

### 4.3 Finding a Witness Segment

The last step of the decision algorithm is to rotate a line along each curve of \(\Gamma(r^*)\). We have \(O(h^2)\) events in total each of which is either a common tangent between two curves of \(\Gamma(r^*)\) or the line tangent to a curve of \(\Gamma(r^*)\) and passing through an endpoint of another curve of \(\Gamma(r^*)\). A common tangent between two curves of \(\Gamma(r)\) is defined by a pair of arcs from two curves. More precisely, it is a common tangent of two arcs of the two curves or a line passing through endpoints of the two curves. Instead of computing the common tangents, we compute the pairs defining them. Since we can compute a common tangent between two convex curves in \(O(\log n)\) time and we have \(O(h^2)\) pairs of convex curves, we can compute all events in \(O(\log n)\) iterations each consisting of \(O(h^2)\) steps. As we did before, we can complete each iteration in \(O(T_p \log(h^2) + h^2) = O(n \log n + h^2 \log^3 h)\) time, where \(T_p\) denotes the running time of the decision algorithm. Therefore, we can obtain \([r_1, r_2]\) such that the set of the pairs of arcs defining the events remains the same for every \(r \in [r_1, r_2]\) in \(O(n \log^2 n + h^2 \log^4 h)\) time. Similarly, we can do this for the events of the other type.

This means that for any \(r \in [r_1, r_2]\), the first curve of \(\Gamma(r)\) hit by \(\ell^+\) remains the same for any line \(\ell\) tangent to a curve of \(\Gamma(r)\). Therefore, the answer of the decision problem remains the same for any \(r \in [r_1, r_2]\). Since \(r^*\) is contained in \([r_1, r_2]\), the answer is positive for every \(r \in [r_1, r_2]\). By definition, we have \(r^* = r_1\). Thus we have the following theorem.

- **Theorem 10.** Given a polygonal domain \(P\) with \(h\) holes of total complexity \(n\), we can compute two points \(s'\) and \(t'\) minimizing \(\max\{d(s, s'), d(t, t')\}\) such that \(s'\) and \(t'\) are visible to each other in \(O(n \log^2 n + h^2 \log^4 h)\) time.
References


Partially Walking a Polygon

Franz Aurenhammer
Institute for Theoretical Computer Science, University of Technology, Graz, Austria
auren@igi.tugraz.at

Michael Steinkogler
Institute for Theoretical Computer Science, University of Technology, Graz, Austria
steinkogler@igi.tugraz.at

Rolf Klein
Universität Bonn, Institut für Informatik, Bonn, Germany
rolf.klein@uni-bonn.de

Abstract
Deciding two-guard walkability of an \( n \)-sided polygon is a well-understood problem. We study the following more general question: How far can two guards reach from a given source vertex while staying mutually visible, in the (more realistic) case that the polygon is not entirely walkable? There can be \( \Theta(n) \) such maximal walks, and we show how to find all of them in \( O(n \log n) \) time.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Polygon, guard walk, visibility

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.60

Funding This work was supported by Project I 1836-N15, Austria Science Fund (FWF).

Acknowledgements We want to thank the anonymous referees for their detailed comments that have helped improving the presentation of this paper.

1 Introduction

We address the following structural question on polygons: How many adjacent ear triangles can be cut off from a polygon \( W \), starting from a given vertex \( s \)? This question was originally motivated by optimizing so-called triangulation axes, a recently introduced skeletal structure for simple polygons [1]. An equivalent formulation of the problem, which is of interest in its own right, reads as follows: How far can two guards reach when they are to walk on \( W \)’s boundary, starting from \( s \) in different directions and staying mutually visible?

Visibility problems of this kind have been studied already in the 1990s, where Icking and Klein [6] gave an \( O(n \log n) \) time algorithm for deciding two-guard walkability of an \( n \)-sided polygon \( W \), from a source vertex \( s \) to a target vertex \( t \). A few years later, Tseng et al. [7] showed that one can find, within the same runtime, all vertex pairs \((s, t)\) such that \( W \) is two-guard walkable from \( s \) to \( t \). Their result was improved to optimal \( O(n) \) time by Bhattacharya et al. [3]. The algorithm in [6] actually provides a walk for \( W \) in case of its existence but, on the other hand, only a negative message is returned in the (quite likely) case that the polygon is not entirely walkable.

The present paper elaborates on ‘how far’ in the latter case a polygon \( W \) is two-guard walkable – a natural question that has not been considered in the literature to the best of our knowledge. Such maximal walks are not unique, in general, which complicates matters. We present a strategy that finds, in \( O(n \log n) \) time, all possible maximal walks that initiate
at a given source vertex $s$ of $W$. A preliminary version of this paper appeared in [2]. For an account of related visibility questions on polygons, we refer to the survey article by Urrutia [8] on art gallery problems.

## 2 Preliminaries

We start with introducing the concepts and notations needed later in our considerations. Throughout, we let $W$ denote a simple polygon in the plane with $n$ vertices, one of them being tagged as a source vertex, $s$. For two points $x$ and $y$ on the boundary, $\partial W$, of $W$, we write $x < y$ if $x$ is reached before $y$ when walking on $\partial W$ from $s$ in clockwise (CW) direction. For a vertex $p$ of $W$, $p^+$ denotes the CW successor vertex of $p$ on $\partial W$. Similarly, $p^-$ denotes the CW predecessor vertex of $p$ on $\partial W$. When $p$ is a reflex vertex (that is, a vertex where the interior angle in $W$ is greater than $\pi$), then the two ‘ray shooting points’ for $p$ in $W$ can be defined, namely, $\text{For}(p)$ as the first intersection point with $\partial W$ of the ray from $p^-$ through $p$, and $\text{Back}(p)$ as the first intersection point with $\partial W$ of the ray from $p^+$ through $p$; consult Figure 1.

According to the aforementioned relation between walks and triangulations, we are only interested in discrete and straight walks. That is, the guards when moving on $\partial W$ directly ‘jump’ from a vertex to the respective neighboring vertex (only one guard is allowed to move at a time), and they never backtrack. A walk in $W$ is now defined as a diagonal $(l, r)$ of $W$, $l < r$, such that the first guard can move CW from $s$ to $l$, and the second guard can move CCW from $s$ to $r$, while staying visible to each other at each step. An obvious condition for $W$ to be walkable till $(l, r)$ is that the two boundary chains from $s$ to $l$ and to $r$, respectively (call them $L$ and $R$), are co-visible in $W$. That is, each vertex on $L$ is visible from some vertex on $R$, and each vertex on $R$ is visible from some vertex on $L$.

To characterize walkability, we will need a few more concepts, first introduced in [6]. We say that $W$ forms a forward deadlock at a pair $(p, q)$ of its reflex vertices if we have

$$\text{Back}(q) < p < q < \text{For}(p).$$

Similarly, $W$ forms a backward deadlock at $(p, q)$ if

$$p < (\text{For}(q), \text{Back}(p)) < q.$$

Finally, $W$ forms a CW wedge at $(p, q)$, if $p < q$ and there exists no vertex $x$ of $W$ with

$$q < \text{For}(q) < x < \text{Back}(p).$$

(A CCW wedge is defined in a symmetric way.) See Figure 1 where these geometric concepts are illustrated.

It is not hard to see that the two guards cannot pass beyond deadlocks and wedges without losing visibility. This will be made specific in Section 4. Moreover, in the work [6] it has been shown that these obstacles to walkability are indeed the only ones. By adapting their result to our setting we get:

**Theorem 1.** Let $(l, r)$, $l < r$, be a diagonal of $W$, and denote with $Q$ the polygon bounded by $(l, r)$ and the two chains $L$ and $R$ defined above. Then $(l, r)$ is a walk in $W$ iff the following three conditions are satisfied:

1. $L$ and $R$ are co-visible in $Q$.
2. $Q$ neither forms a forward deadlock nor a backward deadlock $(p, q)$ with $p \in L$ and $q \in R$, unless $p$ or $q$ is in $(l, r)$,
3. $Q$ forms no CW wedge on $L$, and no CCW wedge on $R$.
3 Extremal walks and obstacles

Given a polygon $W$, our intention is to explore how far $W$ is walkable from the source vertex $s$. That is, we want to find extremal positions for a diagonal $(l, r)$ in $W$ such that $(l, r)$ is still a valid walk. A necessary (but not sufficient) condition is that $(l, r)$ cannot be extended by a single guard move. More adequately, a walk $(l, r)$ in $W$ is termed \textit{maximal} if there is no other walk $(l', r')$ in $W$ such that $l' \geq l$ and $r' \leq r$. For finding maximal walks, we will apply Theorem 1, but we have to do so with care since conditions (1) to (3) refer to a (yet unknown) polygon $Q$, rather than to the input polygon $W$ as in [6].

To this end, for (1) we observe that the chains $L$ and $R$ are co-visible in $Q$ iff they are co-visible in $W$: The line segment $lr$ lies entirely within $W$, so the part of \( \partial W \) different from $\partial Q$ does not obstruct the view within $Q$.

Concerning (2), we notice that forward deadlocks formed by $Q$ do not depend on the shape of $\partial W \setminus \partial Q$, and thus trivially are also forward deadlocks formed by $W$. By contrast, for a backward deadlock $(p, q)$ formed by $Q$, the points $\text{For}(q)$ and $\text{Back}(p)$ in $Q$ may not be the same as in $W$. (Namely, if at least one of them lies on $lr$). But since these points are larger than $p$ and smaller than $q$, $(p, q)$ is also a backward deadlock in $W$.

No such property holds for the wedges in (3), however. A wedge $(p, q)$ formed by $Q$ is not necessarily also formed by $W$: The segment $lr$ can obstruct the view to vertices $x$ on $\partial W \setminus \partial Q$ that prevent $(p, q)$ from being a wedge in $W$. Figure 2 illustrates this situation.

Fortunately though, such ‘induced’ wedges cannot occur as long as the co-visibility condition is satisfied:
Observation 2. Assume that the diagonal $\overline{lr}$ of $W$ induces a wedge in the polygon $Q$ bounded by $\overline{lr}$ and the chains $L$ and $R$. Then $L$ and $R$ are not co-visible.

Proof. Without loss of generality, let the induced wedge, $(p, q)$, be a CW wedge; see Figure 2 again. Then for the reflex vertex $p$ we have $p < \text{Back}(p)$, and because $(p, q)$ is induced by $\overline{lr}$ we also have $r > \text{Back}(p)$. But this implies that the vertex $p^+$ (which belongs to the chain $L$) is not visible from any point on the CCW chain from $p$ to $r$. In particular, $p^+$ is not visible from any vertex on the chain $R$, which ranges from $s$ to $r$.

In summary, we can conclude that it suffices to consider the obstacles formed by the input polygon $W$, rather than the obstacles formed by $Q$.

For maximal walks, obstacles with extremal positions are relevant (in case of the presence of obstacles at all, which we will assume in the sequel). A minimal CW wedge on the chain $L$ is a wedge $(p, q)$ on $L$ where the vertex $q$ is smallest possible. For a minimal CCW wedge $(p, q)$ on $R$, in turn, the vertex $p$ has to be largest possible. Such extremal wedges need not be unique. A representative can be found in $O(n \log n)$ time, by a simple adaption of an algorithm given in [7], which finds all non-redundant wedges of a polygon. (We therefore do not elaborate on the details here.)

A deadlock $(p, q)$ (either forward or backward) is called minimal if there is no other such deadlock $(p', q')$ with $p' < p$ and $q' > q$. The minimal backward deadlock is unique, by the following property:

Observation 3. If $(p, q)$ and $(p', q')$ are two backward deadlocks with $p < p'$ and $q < q'$, then $(p, q')$ is a backward deadlock as well.

To find this minimal deadlock, we simply let $p$ and $q$ run through the reflex vertices of $W$, starting from $s$ in CW and CCW direction, respectively, until the deadlock inequalities for $p$ as well as for $q$ are fulfilled at the same time. This can be done in $O(n)$ time, if $W$ has been preprocessed accordingly in $O(n \log n)$ time using ray shooting; see Chazelle et al. [4].

Minimal forward deadlocks, on the other hand, are not unique in general. This is one of the reasons why maximal walks need not be unique. In fact, $W$ can contain $\Theta(n)$ minimal forward deadlocks $(p_i, q_i)$; see the figure below for $i = 1, 2, 3$. The following algorithm reports all of them. The points on $\partial W$ relevant for this task are the reflex vertices $p_i$ of $W$ plus their ray shooting points $\text{For}(p_i)$. We assume their availability in cyclic order around $W$.

Algorithm MFD

for all relevant points $x$ in CCW order from $s$ do
  if $x = \text{For}(p)$ and $p < x$ then
    Insert $p$ into a CW sorted list $F$
  else if $x$ is a reflex vertex $q$ then
    Search $F$ for the smallest $p$ with $\text{Back}(q) < p$
    if $p$ exists and is unmarked then
      Mark $p$
      Report the forward deadlock $(p, q)$
    end if
  end if
  $x =$ next relevant point
  Delete from $F$ all vertices $p$ with $p \geq x$
In a nutshell, the algorithm scans the boundary of \( W \) in counterclockwise direction, maintaining reflex vertices with forward rayshots on the scanned part of the boundary in a CW sorted list. When a reflex vertex is encountered, this list is used to search for forward deadlocks formed by the current vertex and vertices in the list.

**Lemma 4.** Algorithm MFD reports all minimal forward deadlocks \((p,q)\) in \( W \), and no other pair.

**Proof.** Let \((p,q)\) be a minimal forward deadlock. Then \( q \) is reflex and \( q < \text{For}(p) \) holds. So the list \( F \) contains \( p \) when \( q \) is processed, by the CCW order of processing. Moreover, because \((p,q)\) is minimal, \( p \) is the smallest vertex in \( F \) with \( \text{Back}(q) < q \), and \( p \) is unmarked. Therefore, the algorithm will report \((p,q)\). Conversely, assume that \((p,q)\) gets reported. Then we know \( \text{Back}(q) < p \), and because \( p \) is in \( F \) we know \( q < \text{For}(p) \). Also, \( p < q \) holds by the deletion criterion in the last line. Therefore \((p,q)\) is a forward deadlock.

Concerning minimality, observe first that there cannot be a forward deadlock \((p',q')\) with \( p' < p \) and \( q' \geq q \). Otherwise, \( F \) contains \( p' \) when \((p,q)\) is reported, because we have \( q \leq q' < \text{For}(q') \). Because of \( p' < p \), the algorithm would have reported \((p',q)\) rather than \((p,q)\), or nothing at all if \( p' \) is marked. There also is no forward deadlock \((p',q')\) with \( p' = p \) and \( q' > q \). Otherwise, because of \( q' > q \), \((p',q')\) has been reported already. So \( p' = p \) is marked, and \((p,q)\) does not get reported.

The algorithm can be implemented to run in \( O(n \log n) \) time. It scans \( O(n) \) relevant points, each being processed in constant time apart from the actions on \( F \), which take \( O(n \log n) \) time in total when a balanced search tree for \( F \) is used.

## 4 Constraints from obstacles

Minimal wedges and deadlocks, and also the required co-visibility, give rise to constraints on the vertices \( l \) and \( r \) for a maximal walk \((l,r)\) in the polygon \( W \). We will discuss the constraints on \( l \) in some detail. The situation for \( r \) is symmetric.

We have to distinguish between absolute and conditional constraints. Among the former is the list below. The first two constraints stem from the co-visibility of \( L \) and \( R \), and have been taken from [6]. For the last two constraints, compare Figure 1.

1. For each reflex vertex \( p \) with \( p > \text{For}(p) \): \( l \leq p \).
2. For each reflex vertex \( p \) with \( p < \text{Back}(p) \): \( l \leq \text{Back}(p) \).
3. For the minimal CW wedge \((p,q)\) on \( L \): \( l \leq q \).
4. For the minimal backward deadlock \((p,q)\): \( l \leq p \).

The conditional constraints read as follows:

1. For each \( p \) in (1): If \( r > p \) then \( l < p^- \).
2. For each \( p \) in (2): If \( r > \text{Back}(p) \) then \( l \leq p \).
3. For \((p,q)\) in (3): If \( r > q \) then \( l < q \).

For convenience, we subsume the absolute constraints (1) - (4) into a single one, \( l \leq x \) (where \( x \) is the smallest right-hand side value), and turn it into a conditional constraint:

4. If \( r \geq s \) then \( l \leq x \).

Finally, the minimal forward deadlocks lead to absolute constraints which deserve special attention. Whereas in the case of a backward deadlock \((p,q)\), neither guard can walk beyond these vertices, we have the following observation for the avoidance of a forward deadlock:

**Observation 5.** To avoid the forward deadlock \((p,q)\), only one of the bounds \( l \leq p \) and \( r \geq q \) needs to hold.
Assume now that \( k \) minimal forward deadlocks \((p_1, q_1), \ldots, (p_k, q_k)\) exist, and let the vertices \( p_i \) be sorted in CW order.

\begin{lemma}
Each of the following \( k + 1 \) pairs of bounds for \((l, r)\) avoids all minimal forward deadlocks:
\[(p_1, s), (p_2, q_1), \ldots, (p_k, q_k), (s^-, q_k).\]
\end{lemma}

\begin{proof}
By minimality of the considered deadlocks, we know that the vertices \( q_i \) will be sorted in CW order as well. So, for each index \( i \geq 2 \), Observation 5 tells us that the constraint \( l \leq p_i \) avoids the deadlocks \((p_i, q_i), \ldots, (p_k, q_k)\), and the constraint \( r \geq q_{i-1} \) avoids the remaining deadlocks \((p_1, q_1), \ldots, (p_{i-1}, q_{i-1})\). Moreover, the constraint \( l \leq p_1 \) suffices to avoid all \( k \) deadlocks, and \( r \geq s \) is trivially fulfilled. The same is true for \( r \geq q_k \) and \( l \leq s^- \), respectively.
\end{proof}

In summary, there are \( O(n) \) constraints in total, which can be identified in \( O(n \log n) \) time by the results in Section 3.

\section{Computing all maximal walks}

Section 4 tells us that the goal is to fulfill the constraints in (I) - (IV) simultaneously, though for each of the bounding pairs in Lemma 6 separately. This gives all possible maximal walks – granted the visibility of the reported vertex pairs. But let us come back to the issue of visibility later in this section.

For a fixed bounding pair \((a, b)\), the constraint satisfaction problem can be transformed into the following standard form: For two variables \( l \) and \( r \), with absolute bounds \( a \) and \( b \), respectively, we have two sets of conditional constraints: Namely, a set \( C_L \) containing constraints for \( l \), of the form
\[
r \geq y_i \implies l \leq x_i
\]
and a set \( C_R \) containing constraints for \( r \), of the form
\[
l \leq x_j \implies r \geq y_j
\]
We may assume that all \( x \)-values and \( y \)-values are in \( \{0, 1, \ldots, n\} \). That is, the vertices \( w_0, w_1, \ldots, w_n \) of \( W \), \( w_0 = w_n = s \), are identified with their indices. This is no loss of generality, because only their relative positions (rather than the geometric positions) on \( \partial W \) matter. We want to compute the (unique) maximal pair \((l, r)\) such that
\[
l \leq a, \ r \geq b, \ and \ all \ constraints \ c \in C_L \cup C_R \ are \ fulfilled.
\]

We say that a constraint \( c_i \in C_L \) is active at a value \( r \) if \( r \geq y_i \) holds. Similarly, a constraint \( c_j \in C_R \) is active at \( l \) if we have \( l \leq x_j \). The constraint fulfilling algorithm, CFF, now simply alternates in scanning through the sorted sets \( C_L \) and \( C_R \) (in ascending order of \( y_i \)-values, and in descending order of \( x_j \)-values, respectively), and adjusts the values of \( l \) and \( r \) according to the constraints that become active. In the figure below, active/inactive constraints are indicated with full/dashed arrows.
Algorithm CFF\((a, b, C_L, C_R)\)
\[ l = a, r = b \]
repeat
\[ x = \min\{x_i \mid c_i \in C_L \text{ is active at } r\} \]
\[ l = \min\{l, x\} \]
\[ y = \max\{y_j \mid c_j \in C_R \text{ is active at } l\} \]
\[ r = \max\{r, y\} \]
until \( r = y \) or \( r = b \)
Return the pair \((l, r)\)

Suppose that a function \( \text{VIS}(l, r) \) is available which returns the smallest vertex \( r' \geq r \) such that \((l, r')\) is visible in the polygon \( W \). (That is, \( lr' \) is the first possible diagonal of \( W \) that emanates from vertex \( l \). If \( r' \) does not exist then \( n + 1 \) is returned.) We now present an algorithm that uses \( \text{CFF} \) and \( \text{VIS} \) as subroutines, and is capable of computing, in \( \mathcal{O}(n \log n) \) time, all maximal walks that exist in \( W \). Let \( P = \{(a_1, b_1), \ldots, (a_m, b_m)\} \) be the given set of bounding pairs. We assume that \( a_1, \ldots, a_m \) (and thus \( b_1, \ldots, b_m \)) are in increasing order. In the polygon below, \((l, r)\) and \((l', r')\) are the two possible maximal walks.

Algorithm MAXWALKS\((P, C_L, C_R)\)
\[ l = a_m, r = b_1 \]
\[ r_{\text{rep}} = n + 1 \]
while \( l \geq 0 \) and \( r < r_{\text{rep}} \) do
\[ (l, r) = \text{CFF}(l, r, C_L, C_R) \]
\[ i = \min\{\lambda \mid a_\lambda \geq l\} \]
\[ r_{\text{cand}} = \max\{b_i, r\} \]
\[ r_{\text{vis}} = \text{VIS}(l, r_{\text{cand}}) \]
\[ y = \max\{\varrho \mid \text{all } c \in C_L \text{ active at } \varrho \text{ admit } l\} \]
if \( r_{\text{vis}} \leq \min\{n, y\} \) and \( r_{\text{vis}} < r_{\text{rep}} \) then
\[ \text{Report } (l, r_{\text{vis}}) \]
\[ r_{\text{rep}} = r_{\text{vis}} \]
end if
\[ l = l - 1 \]
end while

Before providing a proof of correctness, we give a short explanation of this algorithm. All the bounding pairs \((a_i, b_i)\) need to fulfill the constraints that are active there, so the algorithm starts by fulfilling the constraints for the vertex pair \((a_m, b_1)\), as these constraints have to be fulfilled in any case. Then the boundary chain of \( W \) from \( a_m \) ‘down to’ \( s \) is scanned in CCW direction, while fulfilling all constraints on both chains. After each constraint fulfillment it is checked whether a candidate vertex pair lies ‘below’ a bounding pair \((a_i, b_i)\), while also ensuring visibility within \( W \) and maximality among walks.

▶ Lemma 7. Algorithm MAXWALKS is correct.
Proof. The value of \( r \) changes only when Algorithm CFF is called, and thus \( r \) cannot decrease. The first call of CFF is with the bounding pair \((a_m,b_1)\), and the subsequent calls are with \((l,r)\) for \( l < a_m \). As soon as we have \( r > b_1 \), some constraint in \( C_R \) is responsible for this. So putting the bound \( r \) for the next call means no additional restriction. This implies that, for all \( l \), we have the equality \( CFF(l,r,C_L,C_R) = CFF(l,b_1,C_L,C_R) \).

We now look at one iteration of the while loop, under the assumption that Algorithm MAXWALKS worked correctly so far. That is, all maximal walks \((l',r')\) with \( l' \geq l \) have been reported, and no other walks. Let \( l_{old} \) be the value of \( l \) before the iteration. Then \((l,r) = CFF(l_{old} - 1,b_1,C_L,C_R)\) holds by the former equality. So we have \((l,r) = CFF(l',b_1,C_L,C_R)\) for \( l_{old} > l' > l \), implying that there is no walk \((l',r')\) for these \( l' \)-values.

There also is no walk \((l,r')\) with \( r' < r_{cand} \), because the bounding pair \((a_i,b_i)\) as well as the constraints in \( C_R \) need to be respected. Concerning \( r_{vis} \), if \( r_{vis} > n \) then no pair \((l,r')\) with \( r' \geq r_{cand} \) is visible, and thus no such pair can be a walk. Further, if \( r_{vis} > y \) then some constraint in \( C_L \) is active at \( r_{vis} \) but does not admit \( l \), so \((l,r_{vis})\) is not a walk either. On the other hand, if \( r_{vis} \leq \min\{n,y\} \) then \((l,r_{vis})\) is a walk, because the pair is visible and fulfills all the constraints. The pair gets reported unless \( r_{vis} \geq r_{rep} \), in which case \((l,r_{vis})\) is not maximal because a larger pair has been reported already.

Turning to runtime considerations now, we can make the following observations. CFF can be implemented such that the bounding pair of the last call is remembered. This way each constraint in \( C_L \cup C_R \) is handled only once: If a call has been with \((l,r)\), the next call will be with \((l',r')\) where \( l' < l \) (and thus \( r' \geq r \)). Thus only \( O(n) \) time is spent in total for all calls to CFF from Algorithm MAXWALKS.

Computing the thresholds \( y \) in MAXWALKS can also be done in total \( O(n) \) time. We remember the previous value of \( y \), and scan down from this value as long as all active constraints of \( C_L \) are fulfilled by \( y \). The first violating constraint then gives the new value for \( y \).

The function VIS can be performed in logarithmic time using the techniques in Guibas and Hershberger [5], in a way similar as already done in Icking and Klein [6]: Basically, finding the desired vertex \( r_{vis} \) can be reduced to finding the first vertex on a shortest path between two polygon vertices. Clearly, the while loop is executed only \( O(n) \) times (because the value of \( l \) is decremented in each iteration), which gives a runtime of \( O(n \log n) \) for this part, and thus for Algorithm MAXWALKS overall.

We now can conclude the main result of this paper:

Theorem 8. Let \( W \) be a simple polygon with \( n \) vertices. For a given vertex \( s \) of \( W \), there can be \( \Theta(n) \) maximal two-guard walks in \( W \) starting from \( s \), and these walks can be computed in \( O(n \log n) \) time.

6 Concluding remarks

A few comments related to the results in this paper are in order.

The polygon example in Algorithm MAXWALKS shows that maximal walks may differ in (combinatorial) length. The walk \((l,r)\) involves 6 steps by the left guard and 3 steps by the right guard, so 9 steps in total, whereas the walk \((l',r')\) involves 8 steps by the left guard and 2 steps by the right guard, and thus allows one more step in total.

The same example also reveals that minimum forward deadlocks are not the only reason why maximal walks are not unique: The reason why there are two walks for the shown
polygon is the vertex \( v \), which can be ‘approached’ by the (mutually visible) guards in two different ways.

In Section 3 we have seen that minimum forward deadlocks can lead to \( \Omega(n) \) different maximal walks. On the other hand, the number of maximal walks trivially cannot exceed \( n \), because no two of them can have the same \( l \)-vertex, or the same \( r \)-vertex, by maximality.

Algorithm MAXWALKS provides each maximal walk in the form of a target pair \((l, r)\), but the algorithm does not specify the way the two guards actually move on \( \partial W \). Such a movement can be computed in \( O(n) \) additional time: Since we know that the subpolygon \( Q \) of \( W \) defined by \( s \) and \((l, r)\) is entirely walkable, we can simply apply the algorithm in [6] to the polygon \( Q \) (which has already been preprocessed with \( W \)).

Notice, however, that a fixed target pair \((l, r)\) may still leave the guards different ways to perform the walk. Different ways to triangulate \( W \) from \( s \) to \((l, r)\) then result. For example, in the polygon example in Algorithm MAXWALKS, it would be possible to include one more diagonal with endpoint \( r \) into the solid-line triangulation, namely, the diagonal \( \tau r \). The dual of any such triangulation has to be a path, though, as the triangulation is constructed by repeatedly cutting off adjacent ear triangles, one triangle per guard step.

References

Abstract

We initiate the study of the following natural geometric optimization problem. The input is a set of axis-aligned rectangles in the plane. The objective is to find a set of horizontal line segments of minimum total length so that every rectangle is stabbed by some line segment. A line segment stab a rectangle if it intersects its left and its right boundary. The problem, which we call STABBING, can be motivated by a resource allocation problem and has applications in geometric network design. To the best of our knowledge, only special cases of this problem have been considered so far.

STABBING is a weighted geometric set cover problem, which we show to be NP-hard. While for general set cover the best possible approximation ratio is $\Theta(\log n)$, it is an important field in geometric approximation algorithms to obtain better ratios for geometric set cover problems. Chan et al. [SODA’12] generalize earlier results by Varadarajan [STOC’10] to obtain sub-logarithmic performances for a broad class of weighted geometric set cover instances that are characterized by having low shallow-cell complexity. The shallow-cell complexity of STABBING instances, however, can be high so that a direct application of the framework of Chan et al. gives only logarithmic bounds. We still achieve a constant-factor approximation by decomposing general instances into what we call laminar instances that have low enough complexity.

Our decomposition technique yields constant-factor approximations also for the variant where rectangles can be stabbed by horizontal and vertical segments and for two further geometric set cover problems.
Stabbing Rectangles by Line Segments

Figure 1 An instance of Stabbing (rectangles) with an optimal solution (gray line segments).

2012 ACM Subject Classification Theory of computation → Packing and covering problems, Theory of computation → Computational geometry

Keywords and phrases Geometric optimization, NP-hard, approximation, shallow-cell complexity, line stabbing

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.61


1 Introduction

In this paper, we study the following geometric optimization problem, which we call Stabbing. The input is a set $R$ of $n$ axis-aligned rectangles in the plane. The objective is to find a set $S$ of horizontal line segments of minimum total length $\|S\|$, where $\|S\| = \sum_{s \in S} \|s\|$, such that each rectangle $r \in R$ is stabbed by some line segment $s \in S$. Here, we say that $s$ stabs $r$ if $s$ intersects the left and the right edge of $r$ (see Fig. 1). The length of a line segment $s$ is denoted by $\|s\|$. Throughout this paper, rectangles are assumed to be axis-aligned and segments are horizontal line segments (unless explicitly stated otherwise).

Our problem can be viewed as a resource allocation problem. Consider a server that receives a number of communication requests. Each request $r$ is specified by a time window $[t_1, t_2]$ and a frequency band $[f_1, f_2]$. In order to satisfy the request $r$, the server has to open a communication channel that is available in the time interval $[t_1, t_2]$ and operates at a fixed frequency within the frequency band $[f_1, f_2]$. Therefore, the server has to open several channels over time so that each request can be fulfilled. Requests may share the same channel if their frequency bands and time windows overlap. Each open channel incurs a fixed cost per time unit and the goal is to minimize the total cost. Consider a $t$–$f$ coordinate system. A request $r$ can be identified with a rectangle $[t_1, t_2] \times [f_1, f_2]$. An open channel corresponds to horizontal line segments and the operation cost equals its length. Satisfying a request is equivalent to stabbing the corresponding rectangle.

To the best of our knowledge, general Stabbing has not been studied, although it is a natural problem. Finke et al. [10] consider the special case of the problem where the left sides of all input rectangles lie on the $y$-axis. They derive the problem from a practical application in the area of batch processing and give a polynomial time algorithm that solves this special case of Stabbing to optimality. Das et al. [6] describe an application of Stabbing in geometric network design. They obtain a constant-factor approximation for a slight generalization of the special case of Finke et al. in which rectangles are only constrained to intersect the $y$-axis. This result constitutes the key step for an $O(\log n)$-approximation algorithm to the Generalized Minimum Manhattan Network problem.

We also consider the following variant of our problem, which we call Constrained Stabbing. Here, the input additionally consists of a set $F$ of horizontal line segments of which any solution $S$ must be a subset.
Related Work. Stabbing can be interpreted as a weighted geometric set cover problem where the rectangles play the role of the elements, the potential line segments correspond to the sets and a segment $s$ “contains” a rectangle $r$ if $s$ stabs $r$. The weight of a segment $s$ equals its length $\|s\|$. Set Cover is one of the classical NP-hard problems. The greedy algorithm yields a $\ln n$-approximation (where $n$ is the number of elements) and this is known to be the best possible approximation ratio for the problem unless $P = NP$ [9, 7]. It is an important research direction of computational geometry to surpass the lower bound known for general Set Cover in geometric settings. In their seminal work, Brönnimann and Goodrich [3] gave an $O(\log \text{OPT})$-approximation algorithm for unweighted Set Cover, where OPT is the size of an optimum solution, for the case when the underlying VC-dimension is constant. This holds in many geometric settings. Numerous subsequent works have improved upon this result in specific geometric settings. For example, Aronov et al. [1] obtained an $O(\log \log \text{OPT})$-approximation algorithm for the problem of piercing a set of axis-aligned rectangles with the minimum number of points (Hitting Set for axis-aligned rectangles) by means of so-called $\varepsilon$-nets. Mustafa and Ray [17] obtained a PTAS for the case of piercing pseudo-disks by points. A limitation of these algorithms is that they only apply to unweighted geometric Set Cover; hence, we cannot apply them directly to our problem. In a break-through, Varadarajan [18] developed a new technique, called quasi-uniform sampling, that gives sub-logarithmic approximation algorithms for a number of weighted geometric set cover problems (such as covering points with weighted fat triangles or weighted disks). Subsequently, Chan et al. [5] generalized Varadarajan’s idea. They showed that quasi-uniform sampling yields a sub-logarithmic performance if the underlying instances have low shallow-cell complexity. Bansal and Pruhs [2] presented an interesting application of Varadarajan’s technique. They reduced a large class of scheduling problems to a particular geometric set cover problem for anchored rectangles and obtained a constant-factor approximation via quasi-uniform sampling. Recently, Chan and Grant [4] and Mustafa et al. [16] settled the APX-hardness status of all natural weighted geometric Set Cover problems where the elements to be covered are points in the plane or space.

Gaur et al. [12] considered the problem of stabbing a set of axis-aligned rectangles by a minimum number of axis-aligned lines. They obtain an elegant 2-approximation algorithm for this NP-hard problem by rounding the standard LP-relaxation. Kovaleva and Spielskma [14] considered a generalization of this problem involving weights and demands. They obtained a constant-factor approximation for the problem. Even et al. [8] considered a capacitated variant of the problem in arbitrary dimension. They obtained approximation ratios that depend linearly on the dimension and extended these results to approximate certain lot-sizing inventory problems. Giannopoulos et al. [13] investigated the fixed-parameter tractability of the problem where given translated copies of an object are to be stabbed by a minimum number of lines (which is also the parameter). Among others, they showed that the problem is $W[1]$-hard for unit-squares but becomes FPT if the squares are disjoint.

Our Contribution. We are the first to investigate Stabbing in this general form: horizontal line segments stabbing axis-aligned rectangles without further restrictions. We examine the complexity and the approximability of this problem.

We rule out the possibility of efficient exact algorithms by showing that Stabbing is NP-hard; see Section 4. Another negative result is that Stabbing instances can have high shallow-cell complexity so that a direct application of the quasi-uniform sampling method yields only the same logarithmic bound as for arbitrary set cover instances; see Section 2.2.
Our main result is a constant-factor approximation algorithm for Stabbing; see Section 2. Our algorithm is based on the following three ideas. First, we show a simple decomposition lemma that implies a constant-factor approximation for (general) set cover instances whose set family can be decomposed into two disjoint sub-families each of which admits a constant-factor approximation. Second, we show that Stabbing instances whose segments have a special laminar structure have low enough shallow-cell complexity so that they admit a constant-factor approximation by quasi-uniform sampling. Third, we show that an arbitrary instance can be transformed in such a way that it can be decomposed into two disjoint laminar families. Together with the decomposition lemma, this establishes the constant-factor approximation.

Another (this time more obvious) application of the decomposition lemma gives also a constant-factor approximation for the variant of Stabbing where we allow horizontal and vertical stabbing segments. Also in this case, a direct application of quasi-uniform sampling gives only a logarithmic bound as there are laminar families of horizontal and vertical segments that have high shallow-cell complexity. This and two further applications of the decomposition lemma are sketched in Section 3.

The above results provide two natural examples for the fact that the property of having low shallow-cell complexity is not closed under the union of the set families. In spite of this, constant-factor approximations are still possible. Our results also show that the representation as a union of low-complexity families may not be obvious at first glance. We therefore hope that our approach helps to extend the reach of quasi-uniform sampling beyond the concept of low shallow-cell complexity also in other settings. Our results for Stabbing may also lead to new insights for other related geometric problems such as the Generalized Minimum Manhattan Network problem [6].

Due to space constraints, we refer the reader for further results such as the APX-hardness of Constrained Stabbing and the relationship of Stabbing to well-studied geometric set cover (or equivalently hitting set) problems to the full version of our paper (see page 2).

2 A Constant-Factor Approximation Algorithm for Stabbing

In this section, we present a constant-factor approximation algorithm for Stabbing. First, we model Stabbing as a set cover problem, and we revisit the standard linear programming relaxation for set cover and the concept of shallow-cell complexity; see Sections 2.1 and 2.2. Then, we observe that there are Stabbing instances with high shallow-cell complexity. This limiting fact prevents us from obtaining any constant approximation factor if applying the generalization of Chan et al. [5] in a direct way; see Section 2.2. In order to bypass this limitation, we decompose any Stabbing instance into two disjoint families of low shallow-cell complexity. Before describing the decomposition in Section 2.5, we show how to merge solutions to these two disjoint families in an approximation-factor preserving way; see Section 2.3. Then, in Section 2.4, we observe that these families have sufficiently small shallow-cell complexity to admit a constant-factor approximation.

2.1 Set Cover and Linear Programming

An instance \((U, F, c)\) of weighted Set Cover is given by a finite universe \(U\) of \(n\) elements, a family \(F\) of subsets of \(U\) that covers \(U\), and a cost function \(c: F \rightarrow \mathbb{Q}^+\). The objective is to find a sub-family \(S\) of \(F\) that also covers \(U\) and minimizes the total cost \(c(S) = \sum_{S \in S} c(S)\).

An instance \((R, F)\) of Constrained Stabbing, given by a set \(R\) of rectangles and a set \(F\) of line segments, can be seen as a special case of weighted Set Cover where the rectangles in \(R\) are the universe \(U\), the line segments in \(F\) form the sets in \(F\), and a line
segment $s \in F$ “covers” a rectangle $r$ if and only if $s$ stabs $r$. Unconstrained Stabbing can be modeled by Set Cover as follows. We can, without loss of generality, consider only feasible solutions where the end points of any line segment lie on the left or right boundaries of rectangles and where each line segment touches the top boundary of some rectangle. Thus, we can restrict ourselves to feasible solutions that are subsets of a set $F$ of $O(n^3)$ candidate line segments. This shows that Stabbing is a special case of Constrained Stabbing and, hence, of Set Cover.

The standard LP relaxation $LP(U, \mathcal{F}, c)$ for a Set Cover instance $(U, \mathcal{F}, c)$ is as follows:

\[
\begin{align*}
\text{Minimize} & \quad \sum_{S \in \mathcal{F}} c(S)z_S \\
\text{subject to} & \quad \sum_{S \in \mathcal{F}, S \ni e} z_S \geq 1 \quad \text{for all } e \in U, \\
& \quad z_S \geq 0 \quad \text{for all } S \in \mathcal{F}.
\end{align*}
\]

The optimum solution to this LP provides a lower bound on $OPT$. An algorithm is called \textit{LP-relative $\alpha$-approximation algorithm} for a class $\Pi$ of set cover instances if it rounds any feasible solution $z = (z_S)_{S \in \mathcal{F}}$ to the above standard LP relaxation for some instance $(U, S, c)$ in this class to a feasible integral solution $S \subseteq \mathcal{F}$ of cost $c(S) \leq \alpha \sum_{S \in \mathcal{F}} c(S)z^*_S$.

### 2.2 Shallow-Cell Complexity

We define the shallow-cell complexity for classes that consist of instances of weighted Set Cover. Informally, the shallow-cell complexity is a bound on the number of equivalent classes of elements that are contained in a small number of sets. Here is the formal definition.

\textbf{Definition 1} (Chan et al. [5]). Let $f(m, k)$ be a function non-decreasing in $m$ and $k$. An instance $(U, \mathcal{F}, c)$ of weighted Set Cover has shallow-cell complexity $f$ if the following holds for every $k$ and $m$ with $1 \leq k \leq m \leq |\mathcal{F}|$, and every sub-family $\mathcal{S} \subseteq \mathcal{F}$ of $m$ sets: All elements that are contained in at most $k$ sets of $\mathcal{S}$ form at most $f(m, k)$ equivalence classes (called \textit{cells}), where two elements are equivalent if they are contained in precisely the same sets of $\mathcal{S}$. A class of instances of weighted Set Cover has shallow-cell complexity $f$ if all its instances have shallow-cell complexity $f$.

Chan et al. proved that if a set cover problem has low shallow-cell complexity then quasi-uniform sampling yields an LP-relative approximation algorithm with good performance.

\textbf{Theorem 2} (Chan et al. [5]). Let $\varphi(m)$ be a non-decreasing function, and let $\Pi$ be a class of instances of weighted Set Cover. If $\Pi$ has shallow-cell complexity $m\varphi(m)k^{O(1)}$, then $\Pi$ admits an LP-relative approximation algorithm (based on quasi-uniform sampling) with approximation ratio $O(\max\{1, \log \varphi(m)\})$.

Unfortunately, there are instances of Stabbing (and its constrained variants) that have high shallow-cell complexity, so we cannot directly obtain a sub-logarithmic performance via Theorem 2. These instances can be constructed as follows; see Fig. 2a. Let $m$ be an even positive integer. For $i = 1, \ldots, m$, define the point $p_i = (i, i)$. For each pair $i, j$ with $1 \leq i \leq m/2 < j \leq m$, let $r_{ij}$ be the rectangle with corners $p_i$ and $p_j$. Now, consider the following set $\mathcal{S}$ of $m$ line segments. For $i = 1, \ldots, m/2$, the set $\mathcal{S}$ contains the segment $s_i$ with endpoints $p_i$ and $(m, i)$. For $i = m/2 + 1, \ldots, m$, the set $\mathcal{S}$ contains the segment $s_i$ with endpoints $(1, i)$ and $p_i$. We want to count the number of rectangles that are stabbed by at most two segments in $\mathcal{S}$. Consider any $i$ and $j$ satisfying $1 \leq i \leq m/2 < j \leq m$. Observe that
the rectangle \( r_{ij} \) is stabbed precisely by the segments \( s_i \) and \( s_j \) in \( \mathcal{S} \). Hence, according to Definition 1, our instance consists of at least \( m^2/4 \) equivalence classes for \( k = 2 \). Thus, if our instance has shallow cell-complexity \( f \) for some suitable function \( f \), we have \( f(m, 2) = \Omega(m^2) \). Since \( f \) is non-decreasing, we also have \( f(m, k) = \Omega(m^2) \) for \( k \geq 2 \). Hence, Theorem 2 implies only an \( O(\log n) \)-approximation algorithm for \textsc{Stabbing} (and its constrained variants) where we use the above-mentioned fact (see Section 2.1) that we can restrict ourselves to \( m = O(n^2) \) many candidate segments.

2.3 Decomposition Lemma for Set Cover

Our trick is to decompose general instances of \textsc{Stabbing} (which may have high shallow-cell complexity) into partial instances of low complexity with a special, laminar structure. We use the following simple decomposition lemma, which holds for arbitrary set cover instances.

\begin{lemma}
Let \( \Pi, \Pi_1, \Pi_2 \) be classes of \textsc{Set Cover} where \( \Pi_1 \) and \( \Pi_2 \) admit LP-relative \( \alpha_1 \)- and \( \alpha_2 \)-approximation algorithms, respectively. The class \( \Pi \) admits an LP-relative \( (\alpha_1 + \alpha_2) \)-approximation algorithm if, for every instance \( (U, \mathcal{F}, c) \in \Pi \), the family \( \mathcal{F} \) can be partitioned into \( \mathcal{F}_1, \mathcal{F}_2 \) such that, for any partition of \( U \) into \( U_1, U_2 \) where \( U_1 \) is covered by \( \mathcal{F}_1 \) and \( U_2 \) by \( \mathcal{F}_2 \), the instances \( (U_1, \mathcal{F}_1, c) \) and \( (U_2, \mathcal{F}_2, c) \) are instances of \( \Pi_1 \) and \( \Pi_2 \), respectively.
\end{lemma}

\begin{proof}
Let \( z = (z_S)_{S \in \mathcal{F}} \) be a feasible solution to LP\((U, \mathcal{F}, c)\). Let \( U_1, U_2 = \emptyset \) initially. Consider an element \( e \in U \). Because of the constraint \( \sum_{S \in \mathcal{F}_1, S \ni e} z_S \geq 1 \) in the LP relaxation and because of \( \mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2 \), at least one of the cases \( \sum_{S \in \mathcal{F}_1, S \ni e} z_S \geq \alpha_1 / (\alpha_1 + \alpha_2) \) and \( \sum_{S \in \mathcal{F}_2, S \ni e} z_S \geq \alpha_2 / (\alpha_1 + \alpha_2) \) occurs. If the first case holds, we add \( e \) to \( U_1 \). Otherwise, the second case holds and we add \( e \) to \( U_2 \). We execute this step for each element \( e \in U \).

Now, consider the instance \( (U_1, \mathcal{F}_1, c) \). For each \( S \in \mathcal{F}_1 \), set \( z_S^1 := \min\{z_S(\alpha_1 + \alpha_2) / \alpha_1, 1\} \). Since \( \sum_{S \in \mathcal{F}_1, S \ni e} z_S \geq 1 / (\alpha_1 + \alpha_2) \) for all \( e \in U_1 \), we have that \( z^1 = (z_S^1)_{S \in \mathcal{F}_1} \) forms a feasible solution to LP\((U_1, \mathcal{F}_1, c)\). Next, we apply the LP-relative \( \alpha_1 \)-approximation algorithm to this instance to obtain a solution \( S_1 \subseteq \mathcal{F}_1 \) that covers \( U_1 \) and whose cost is at most \( \alpha_1 \sum_{S \in \mathcal{F}_1} c(S) z_S^1 \leq (\alpha_1 + \alpha_2) \sum_{S \in \mathcal{F}_1} c(S) z_S \). Analogously, we can compute a solution \( S_2 \subseteq \mathcal{F}_2 \) to \( (U_2, \mathcal{F}_2, c) \) of cost at most \( (\alpha_1 + \alpha_2) \sum_{S \in \mathcal{F}_2} c(S) z_S \).

To complete the proof, note that \( S_1 \cup S_2 \) is a feasible solution to \( (U, \mathcal{F}, c) \) of cost at most \( (\alpha_1 + \alpha_2) \sum_{S \in \mathcal{F}_1 \cup \mathcal{F}_2} c(S) z_S \). Hence, our algorithm is an LP-relative \( (\alpha_1 + \alpha_2) \)-approximation algorithm.
\end{proof}

2.4 \textit{x}-Laminar Instances

\begin{definition}
An instance of \textsc{Constrained Stabbing} is called \textit{x-laminar} if the projection of the segments in this instance onto the \( x \)-axis forms a laminar family of intervals. That is, any two of these intervals are either interior-disjoint or one is contained in the other.
\end{definition}
We remark that for an $x$-laminar instance of \textsc{Constrained Stabbing} the corresponding instance $(U, F, c)$ of \textsc{Set Cover} does not necessarily have a laminar set family $F$.

\textbf{Lemma 5.} The shallow-cell complexity of an $x$-laminar instance of \textsc{Constrained Stabbing} can be upper bounded by $f(m, k) = mk^2$. Hence, such instances admit a constant-factor LP-relative approximation algorithm.

\textbf{Proof.} To prove the bound on the shallow-cell complexity, consider a set $S$ of $m$ segments. Let $1 \leq k \leq m$ be an integer. Consider an arbitrary rectangle $r$ that is stabbed by at most $k$ segments in $S$. Let $S_r$ be the set of these segments. Consider a shortest segment $s \in S_r$. By laminarity, the projection of any segment in $S_r$ onto the $x$-axis contains the projection of $s$ onto the $x$-axis. Let $C_s = (s_1, \ldots, s_k)$ be the sequence of all segments in $S$ whose projection contains the projection of $s$, ordered from top to bottom. The crucial point is that the set $S_r$ forms a contiguous sub-sequence $s_i, \ldots, s_{i+|S_r|-1}$ of $C_s$ that contains $s = s_j$ for some $i \leq j \leq i + |S_r| - 1$. Hence, $S_r$ is uniquely determined by the choice of $s \in S$ (for which there are $m$ possibilities), the choice of $s_j$ with $i \in \{j-k, \ldots, j\}$ within the sequence $C_s$ (for which there are at most $k$ possibilities), and the cardinality of $S_r$ (for which there are at most $k$ possibilities). This implies that $S_r$ is one of $mk^2$ many sets that define a cell. This completes our proof since $r$ was picked arbitrarily.

\subsection{ Decomposing General Instances into Laminar Instances }

\textbf{Lemma 6.} Given an instance $I$ of (unconstrained) \textsc{Stabbing} with rectangle set $R$, we can compute an instance $I' = (R, F)$ of \textsc{Constrained Stabbing} with the following properties. The set $F$ of segments in $I'$ has cardinality $O(n^3)$, it can be decomposed into two disjoint $x$-laminar sets $F_1$ and $F_2$, and $\text{OPT}_{I'} \leq 6 \cdot \text{OPT}_I$.

\textbf{Proof.} Let $F'$ be the set of $O(n^3)$ candidate segments as defined in Sec. 2.1: For every segment $s$ of $F'$, the left endpoint of $s$ lies on the left boundary of some rectangle, the right endpoint of $s$ lies on the right boundary of some rectangle, and $s$ contains the top boundary of some rectangle. Recall that $F'$ contains the optimum solution.

Below, we stretch each of the segments in $F'$ by a factor of at most 6 to arrive at a set $F$ of segments having the claimed properties. By scaling the instance we may assume that the longest segment in $F'$ has length $1/3$.

For any $i, j \in \mathbb{Z}$ with $i \geq 0$, let $I_{ij}$ be the interval $[j/2^i, (j+1)/2^i]$. Let $I_1$ be the family of all such intervals $I_{ij}$. We say that $I_{ij}$ has level $i$. Note that $I_1$ is an $x$-laminar family of intervals (segments). Let $I_2$ be the family of intervals that arises if each interval in $I_1$ is shifted to the right by the amount of $1/3$. That is, $I_2$ is the family of all intervals of the form $I_{ij} + 1/3 := [j/2^i + 1/3, (j+1)/2^i + 1/3]$ (for any $i, j \in \mathbb{Z}$ with $i \geq 0$). Clearly, $I_2$ is $x$-laminar, too.

We claim that any arbitrary interval $J = [a, b]$ of length at most $1/3$ is contained in an interval $I$ that is at most 6 times longer than $J$ and that is contained in $I_1$ or in $I_2$. This completes the proof of the lemma since then any segment in $F'$ can be stretched by a factor of at most 6 so that its projection on the $x$-axis lies in $I_1$ (giving rise to the segment set $F_1$) or in $I_2$ (giving rise to the segment set $F_2$). Setting $F = F_1 \cup F_2$ completes the construction of the instance $I' = (R, F)$.

To show the above claim, let $s$ be the largest non-negative integer with $b - a \leq 1/(3 \cdot 2^s)$. If $J$ is contained in the interval $I_{s,j}$ for some integer $j$, we are done because $b - a \leq 1/(6 \cdot 2^s)$ by the choice of $s$. If $J$ is not contained in any interval $I_{s,j}$, then there exists some integer $j$ such that $j/2^s \in J = [a, b]$ and thus $a \in I_{s,j-1}$. Since $b - a \leq 1/(3 \cdot 2^s)$, we have that $J$ is completely contained in the interval $I' := I_{s,j-1} + 1/(3 \cdot 2^s)$ and in the interval $I'' := I_{s,j} - 1/(3 \cdot 2^s)$.\footnote{Note: This is a truncated version of the text. The full version is longer and includes more details and proofs.}
We complete the proof by showing that one of the intervals $I', I''$ is actually contained in $I_2$. To this end, note that $1/3 = \sum_{\ell=1}^{\infty} (-1)^{\ell-1}/2^{\ell}$. Hence, if $s$ is even, the interval $I' - 1/3$ lies in $I_1$, and if $s$ is odd, the interval $I'' - 1/3$ lies in $I_1$.

Applying the decomposition lemma to Lemmas 5 and 6 yields our main result. We do not give an explicit approximation factor due to our reliance on the result by Chan et al. [5]. We also cannot apply a decomposition technique similar to Constrained Stabbing since Lemma 6 requires a free choice of the set $F$ of stabbing line segments.

**Theorem 7.** Stabbing admits a constant-factor LP-relative approximation algorithm.

Complementing Lemmas 5 and 6, Fig. 2a shows that the union of two $x$-laminar families of segments may have shallow-cell complexity with quadratic dependence on $m$. Hence, the property of having low shallow-cell complexity is not closed under taking unions.

### 3 Further Applications of the Decomposition Lemma

Here we show that our decomposition technique can be applied in other settings, too.

**Horizontal–Vertical Stabbing.** In this new variant of Stabbing, a rectangle may be stabbed by a horizontal or by a vertical line segment (or by both). Using the results of Section 2.5 and the decomposition lemma where we decompose into horizontal and vertical segments, we immediately obtain the following result.

**Corollary 8.** Horizontal–Vertical Stabbing admits an LP-relative constant-factor approximation algorithm.

Figure 2b shows that a laminar family of horizontal segments and vertical segments may have a shallow-cell complexity with quadratic dependence on $m$. Thus, Corollary 8 is another natural example where low shallow-cell complexity is not closed under union and where the decomposition lemma gives a constant-factor approximation although the shallow-cell complexity is high.

**Stabbing 3D-Boxes by Squares.** In the 3D-variant of Stabbing, we want to stab 3D-boxes with axis-aligned squares, minimizing the sum of the areas or the sum of the perimeters of the squares. Here, “stabbing” means “completely cutting across”. By combining the same idea with shifted quadtrees – the 2D-equivalent of laminar families of intervals – we obtain a constant-factor approximation for this problem. It is an interesting question if our approach can be extended to handle also arbitrary rectangles but this seems to require further ideas.

**Covering Points by Anchored Squares.** Given a set $P$ of points that need to be covered and a set $A$ of anchor points, we want to find a set of axis-aligned squares such that each square contains at least one anchor point, the union of the squares covers $P$, and the total area or the total perimeter of the squares is minimized. Again, with the help of shifted quadtrees, we can apply the decomposition lemma. In this case, we do not even need to apply the machinery of quasi-uniform sampling; instead, we can use dynamic programming on the decomposed instances. This yields a deterministic algorithm with a concrete constant approximation ratio ($4 \cdot 6^2$, without polishing).
Figure 3 Obtaining a visibility representation from a Planar Vertex Cover instance.

Figure 4 The vertex gadget $R_v$ of vertex $v$.

4 NP-Hardness of Stabbing

To show that Stabbing is NP-hard, we reduce from Planar Vertex Cover: Given a planar graph $G$ and an integer $k$, decide whether $G$ has a vertex cover of size at most $k$. This problem is NP-hard [11]. Omitted proofs can be found in the full version of the paper.

Theorem 9. Stabbing is NP-hard, even for interior-disjoint rectangles.

Let $G = (V, E)$ be a planar graph with $n$ vertices, and let $k$ be a positive integer. Our reduction will map $G$ to a set $R$ of rectangles and $k$ to another integer $k^*$ such that $(G, k)$ is a yes-instance of Planar Vertex Cover if and only if $(R, k^*)$ is a yes-instance of Stabbing. Consider a visibility representation of $G$, which represents the vertices of $G$ by non-overlapping vertical line segments (called vertex segments), and each edge of $G$ by a horizontal line segment (called edge segment) that touches the vertex segments of its endpoints; see Figs. 3a and 3b. Any planar graph admits a visibility representation on a grid of size $O(n) \times O(n)$, which can be found in polynomial time [15]. We compute such a visibility representation for $G$. Then we stretch the vertex segments and vertically shift the edge segments so that no two edge segments coincide (on a vertex segment); see Fig. 3c. The height of the visibility representation remains linear in $n$.

In the next step, we create a Stabbing instance based on this visibility representation, using the edge segments and vertex segments as indication for where to put our rectangles. All rectangles will be interior-disjoint, have positive area and lie on an integer grid that we obtain by scaling the visibility representation by a sufficiently large factor (linear in $n$). A vertex segment will intersect $O(n)$ rectangles (lying above each other since they are disjoint), and each rectangle will have width $O(n)$. The precise number of rectangles and their sizes will depend on the constraints formulated below. Our construction will be polynomial in $n$.

For each edge $e$ in $G$, we introduce an edge gadget $r_e$, which is a rectangle that we place such that it is stabbed by the edge segment of $e$ in the visibility representation.
For each vertex $v$ in $G$, we introduce a vertex gadget $R_v$ as shown in Fig. 4a. It consists of an odd number of rectangles that are (vertically) stabbed by the vertex segment of $v$ in the visibility representation. Any two neighboring rectangles share a horizontal line segment. Its length is exactly $n + 3$ if neither of the rectangles is the top-most rectangle $r_{\text{top}}$ or the bottom-most rectangle $r_{\text{bot}}$. Otherwise, the intersection length equals the width of the respective rectangle $r_{\text{top}}$ or $r_{\text{bot}}$. We set the widths of $r_{\text{top}}$ and $r_{\text{bot}}$ to 1 and 2, respectively.

A vertex gadget $R_v$ is called incident to an edge gadget $e$ if $v$ is incident to $e$.

Before we describe the gadgets and their relation to each other in more detail, we construct, in two steps, a set $S^v$ of line segments for each vertex gadget $R_v$. First, let $S^v$ be the set of line segments that correspond to the top and bottom edges of the rectangles in $R_v$. Second, replace each pair of overlapping line segments in $S^v$ by its union. Then number the line segments in $S^v$ from top to bottom starting with 1. Let $S^v_{\text{ina}}$ be the set of the odd-numbered line segments, and let $S^v_{\text{act}}$ be the set of the even-numbered ones; see Figs. 4b and 4c. By construction, $S^v_{\text{act}}$ and $S^v_{\text{ina}}$ are feasible stabbings for $R_v$. Furthermore, $|S^v_{\text{ina}}| = |S^v_{\text{act}}|$ as $|R_v|$ is odd and, hence, $|S^v|$ is even. Given the difference in the widths of $r_{\text{top}}$ and $r_{\text{bot}}$, we have that $\|S^v_{\text{act}}\| = \|S^v_{\text{ina}}\| + 1$. Note that this equation holds regardless of the widths of the rectangles in $R_v \setminus \{r_{\text{top}}, r_{\text{bot}}\}$.

The rectangles of all gadgets together form a STABBING instance $R$. They meet two further constraints: First, no two rectangles of different vertex gadgets intersect. We can achieve this by scaling the visibility representation by an appropriate factor linear in $n$. Second, each edge gadget $e$ intersects exactly two rectangles, one of its incident left vertex gadgets, $R_u$, and one of its incident right vertex gadgets, $R_v$. The top edge of $e$ touches a segment of $S^u_{\text{act}}$ and the bottom edge of $e$ touches a segment of $S^v_{\text{act}}$. The length of each of the two intersections is exactly $n + 3$; see Fig. 5. Thus, we have $|R_v| = O(\deg(v)) = O(n)$.

Let $S$ be a feasible solution to the instance $R$. We call a vertex gadget $R_v$ active in $S$ if $\{s \cap \bigcup R_e \mid s \in S\} = S^v_{\text{act}}$, and inactive in $S$ if $\{s \cap \bigcup R_e \mid s \in S\} = S^v_{\text{ina}}$. We will see that in any optimum solution each vertex gadget is either active or inactive. Furthermore, we will establish a direct correspondence between the PLANAR VERTEX COVER instance $G$ and the STABBING instance $R$: Every optimum solution to $R$ covers each edge gadget by an active vertex gadget while minimizing the number of active vertex gadgets.

Let $\OPT_G$ denote the size of a minimum vertex cover for $G$, let $\OPT_R$ denote the length of an optimum solution to $R$, let $\text{width}(r)$ denote the width of a rectangle $r$, and finally let $c = \sum_{e \in E} (\text{width}(r_e) - n - 3) + \sum_{v \in V} \|S^v_{\text{ina}}\|$. To show NP-hardness of STABBING, we prove that $\OPT_G \leq k$ if and only if $\OPT_R \leq c + k$. We show the two directions separately.

\begin{lemma}
$\OPT_G \leq k$ implies that $\OPT_R \leq c + k$.
\end{lemma}

\textbf{Proof sketch.} Set each vertex gadget to active if it corresponds to a vertex in the given vertex cover, otherwise to inactive. Stab each edge gadget by prolonging one of the line segments that it touches. Using $\|S^v_{\text{act}}\| = \|S^v_{\text{ina}}\| + 1$, the bound follows.
Next we show the other, more challenging direction. Consider an optimum solution $S_{\text{OPT}}$ to $R$ and choose $k \leq n$ such that $OPT_R \leq c + k$ is satisfied. Let $R_v$ be any vertex gadget, let $r_{\text{top}}$ and $r_{\text{bot}}$ be its top- and bottom-most rectangles, respectively, and let $S_{\text{OPT}} = \{s \cap \bigcup R_v \mid s \in S_{\text{OPT}}\}$. In the following, we prove that $S_{\text{OPT}}^u$ equals either $S^u_{\text{ina}}$ or $S^u_{\text{act}}$.

\begin{itemize}
  \item \textbf{Lemma 11.} If $S^u_{\text{ina}} \not\subseteq S_{\text{OPT}}^u$ and $S_{\text{act}}^u \not\subseteq S_{\text{OPT}}^u$, then $\|S_{\text{OPT}}^u\| > \|S_{\text{ina}}^u\| + n$.
\end{itemize}

\textbf{Proof sketch.} Consider all pairs of neighboring rectangles in $R_v$ that are stabbed by the same line segment of $S_{\text{OPT}}^u$. Let $P$ be a maximum-cardinality subset of these pairs such that every rectangle appears at most once. Thus, $\sum_{r \in R_v} \text{width}(r) - \sum_{(r_1, r_2) \in P} \text{width}(r_1 \cap r_2)$ is a lower bound of $\|S_{\text{OPT}}^u\|$. Observe that the lower bound is minimized if the total intersection length of the rectangles in $P$ is maximized. This happens (even with tightness) if and only if $S_{\text{OPT}}^u = S^u_{\text{ina}}$. Given that $|R_v|$ is odd, there is at least one rectangle not in $P$. If $S^u_{\text{ina}} \not\subseteq S_{\text{OPT}}^u$ and $S_{\text{act}}^u \not\subseteq S_{\text{OPT}}^u$, there is a rectangle $r$ not in $P$ that is neither $r_{\text{top}}$, $r_{\text{bot}}$ nor a neighbor of those. Thus, $r$ contributes $n + 3$ to the total intersection length in $S^u_{\text{ina}}$ but nothing in $S_{\text{OPT}}$. The difference of the total intersection lengths implies the lemma.

\begin{itemize}
  \item \textbf{Lemma 12.} Exactly one of the following three statements holds:
    \begin{enumerate}
      \item $S_{\text{OPT}}^u = S^u_{\text{ina}}$, or
      \item $S_{\text{OPT}}^u = S^u_{\text{act}}$, or
      \item $\|S_{\text{OPT}}^u\| > \|S^u_{\text{ina}}\| + n$.
    \end{enumerate}
\end{itemize}

\textbf{Proof sketch.} If $S^u_{\text{ina}} \not\subseteq S^u_{\text{OPT}}$, there is a line segment $s \in S^u_{\text{OPT}} \setminus S^u_{\text{ina}}$ that stab a rectangle in $R_v \setminus \{r_{\text{top}}, r_{\text{bot}}\}$. By construction, its length is at least $n + 3$. Hence, $\|S^u_{\text{OPT}}\| > \|S^u_{\text{ina}}\| + n$. The same holds if $S^u_{\text{act}} \not\subseteq S^u_{\text{OPT}}$.

Now, we show that $S_{\text{OPT}}$ forces each vertex gadget to be either active or inactive.

\begin{itemize}
  \item \textbf{Lemma 13.} In $S_{\text{OPT}}$, each vertex gadget is either active or inactive.
\end{itemize}

\textbf{Proof.} Suppose that there is a vertex gadget $R_u$ that is neither active nor inactive in $S_{\text{OPT}}$. This implies $OPT_R > c + n$ and contradicts our previous assumption $OPT_R \leq c + k \leq c + n$.

To this end, we give a lower bound on $OPT_R$. Since $R_u$ is neither active nor inactive, $OPT_R > \|S^u_{\text{ina}}\| + n$ by Lemma 12. Thus, $\sum_{v \in V} \|S_{\text{OPT}}^v\| > \sum_{v \in V} \|S^u_{\text{ina}}\| + n$. Let $S_{\text{OPT}}^u$ be the set of all segment fragments of $S_{\text{OPT}}$ lying outside of $\bigcup_{v \in V} S^u_{\text{OPT}}$. Each edge gadget $r_e$ contains a segment fragment from $S^u_{\text{OPT}}$ of length at least width($r_e$) $- n - 3$ since, by construction, it can share a line segment with only one of its incident vertex gadgets. Since all edge gadgets are interior-disjoint, we have $\|S^u_{\text{OPT}}\| \geq \sum_{e \in E} \text{width}(r_e) - n - 3$. Hence,

\[ OPT_R \geq \|S^u_{\text{OPT}}\| + \sum_{v \in V} \|S_{\text{OPT}}^v\| > \sum_{e \in E} \text{width}(r_e) - n - 3 + \sum_{v \in V} \|S^u_{\text{ina}}\| + n = c + n. \]
Given $S_{OPT}$, we put exactly those vertices in the vertex cover whose vertex gadgets are active. By Lemma 14, this yields a vertex cover of $G$. By Lemma 15, the size of the vertex cover is exactly $OPT_R - c$, which is bounded from above by $k$ given that $OPT_R \leq c + k$.

**Lemma 16.** $OPT_R \leq c + k$ implies that $OPT_G \leq k$.

By our construction, we represent $R$ on a grid of size polynomial in $n$, hence, all numerical values are upperbounded by a polynomial in $n$. Our construction is polynomial. With Lemmas 10 and 16, we conclude that Stabbing is NP-hard.

References


Impatient Online Matching

Xingwu Liu
SKL Computer Architecture, ICT, CAS
University of Chinese Academy of Sciences, Beijing, China
liuxingwu@ict.ac.cn

Zhida Pan
Institute of Computing Technology, Chinese Academy of Sciences
University of Chinese Academy of Sciences, Beijing, China
zhidapan@gmail.com

Yuyi Wang
ETH Zurich, Switzerland
yuwang@ethz.ch

Roger Wattenhofer
ETH Zurich, Switzerland
wattenhofer@ethz.ch

Abstract
We investigate the problem of Min-cost Perfect Matching with Delays (MPMD) in which requests are pairwise matched in an online fashion with the objective to minimize the sum of space cost and time cost. Though linear-MPMD (i.e., time cost is linear in delay) has been thoroughly studied in the literature, it does not well model impatient requests that are common in practice. Thus, we propose convex-MPMD where time cost functions are convex, capturing the situation where time cost increases faster and faster. Since the existing algorithms for linear-MPMD are not competitive any more, we devise a new deterministic algorithm for convex-MPMD problems. For a large class of convex time cost functions, our algorithm achieves a competitive ratio of $O(k)$ on any $k$-point uniform metric space. Moreover, our deterministic algorithm is asymptotically optimal, which uncover a substantial difference between convex-MPMD and linear-MPMD which allows a deterministic algorithm with constant competitive ratio on any uniform metric space.

2012 ACM Subject Classification Theory of computation → Online algorithms

Keywords and phrases online algorithm, online matching, convex function, competitive analysis, lower bound

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.62

Acknowledgements This work is partially supported by the National Key Research and Development Program of China (Grant No. 2016YFB1000200), the National Natural Science Foundation of China (61420106013), State Key Laboratory of Computer Architecture Open Fund (CARCH3410).

1 Introduction
Online matching has been studied frantically in the last years. Emek et al. [10] started the renaissance by introducing delays and optimizing the trade-off between timeliness and quality of the matching. This new paradigm leads to the problem of Min-cost Perfect Matching with Delays (MPMD for short), where requests arrive in an online fashion and need to be matched with one another up to delays. Any solution experiences two kinds of costs or penalty. One
Impatient Online Matching

is for quality: Matching two requests of different types incurs cost as such do not match well, while requests of the same type should be matched for free. The other is for timeliness: Delay in matching a request causes a cost that is an increasing function, called the time cost function, of the waiting time. The overall objective is to minimize the sum of the two kinds of costs.

Tractable in theory and fascinating in practice, the MPMD problem has attracted more and more attention and inspired an increasing volume of literature [10, 11, 4, 3, 2]. However, these existing work in this line only studied linear time cost function, meaning that penalty grows at a constant rate no matter how long the delay is. This sharply contrasts to much of our real-life experience. Just imagine a dinner guest: waiting a short time is no problem – but eventually, every additional minute becomes more annoying than ever. The discontentment is experiencing convex growth, an omnipresent concept in biology, physics, engineering, or economics.

Actually, such convex growth of discontentment appears in various real-life scenarios of online matching. For instance, online game platforms often have to match pairs of players before starting a game (consider chess as an example). Players at the same, or at least similar, level of skills should be paired up so as to make a balanced game possible. Then it would be better to delay matching a player in case of no ideal candidate of opponents. Usually it is acceptable that a player waits for a short time, but a long delay may be more and more frustrating and even make players reluctant to join the platform again. Another example appears in organ transplantation: An organ transplantation recipient may be able to wait a bit, but waiting an extended time will heavily affect its health. One may think that organ transplantation would be better modeled by bipartite matching rather than regular matching as considered in this paper; however, organ-recipients and -donors usually come in incompatible pairs that will be matched with other pairs, e.g., two-way kidney exchange1. More real-life examples include ride sharing (match two customers), joint lease (match two roommates), just mention a few.

On this ground, we study the convex-MPMD problem, i.e., the MPMD problem with convex time cost functions. To the best of our knowledge, this is the first work on online matching with non-linear time cost.

Convexity of the time cost poses special challenges to the MPMD problem. An important technique in solving linear-MPMD, namely, MPMD with linear time cost function, is to minimize the total costs while sacrifice some requests by possibly delaying them for a long period (see, e.g., the algorithms in [4, 11, 2]). Because the time cost increases at a constant rate, it is the total waiting time, rather than waiting time of individual requests, that is of interest. Hence, keeping a request waiting is not too harmful. The case of convex time costs is completely different, since we cannot afford anymore to delay old unmatched requests, as their time costs grow faster and faster. Instead, early requests must be matched early. For this reason, existing algorithms for the linear-MPMD problem do not work any more for convex-MPMD, as confirmed by examples in Section 4.

In this paper, we devise a novel algorithm $A$ for the convex-MPMD problem which is deterministic and solves the problem optimally. More importantly, our results disclose a separation: the convex-MPMD problem, even when the cost function is just a little different from linear, is strictly harder than its linear counterpart. Specifically, our main results are as follows, where $f$-MPMD stands for the MPMD problem with time cost function $f$:

Theorem 1. For any \( f(t) = t^\alpha \) with constant \( \alpha > 1 \), the competitive ratio of \( A \) for \( f\text{-MPMD} \) on \( k \)-point uniform metric space is \( O(k) \).

One may wonder whether the result in Theorem 1 can be further improved because of the known result:

Theorem 2 ([4, 2]). There exists a deterministic online algorithm that solves linear-MPMD on uniform metrics and reaches an \( O(1) \) competitive ratio.

However, we can show that for a large family of functions \( f : \mathbb{R}^+ \rightarrow \mathbb{R}^+ \), the \( f\text{-MPMD} \) problem has no deterministic algorithms of competitive ratio \( o(k) \).

Theorem 3. Suppose that the time cost function \( f \) is nondecreasing, unbounded, continuous and satisfies \( f(0) = f'(0) = 0 \). Then any deterministic algorithm for \( f\text{-MPMD} \) on \( k \)-point uniform metric space has competitive ratio \( \Omega(k) \).

Numerous natural convex functions over the domain of nonnegative real numbers satisfy the conditions of Theorem 3. Examples include monomial \( f(t) = t^\alpha \) with \( \alpha > 1 \), \( f(t) = e^{\alpha t} - \alpha t - 1 \) with \( \alpha > 1 \), and so on. This, together with Theorem 1, establishes the optimality of our deterministic algorithm. Note that family of functions satisfying the conditions of Theorem 3 is closed under multiplication and linear combination where the coefficients are positive. Hence, Theorem 3 is of general significance.

2 Related Work

Matching has become one of the most extensively studied problems in graph theory and computer science since the seminal work of Edmonds [9, 8]. Karp et al. [15] studied the matching problem in the context of online computation which inspired a number of different versions of online matching, e.g., [13, 16, 18, 19, 6, 12, 1, 7, 17, 20, 21]. In these online matching problems, underlying graphs are assumed bipartite and requests of one side are given in advance.

A matching problem where all requests arrive in an online manner was introduced by [10]. This paper also introduced the idea that requests are allowed to be matched with delays that need to be paid as well, so the problem is called Min-cost Perfect Matching with Delays (MPMD). They presented a randomized algorithm with competitive ratio \( O(\log^2 k + \log \Delta) \) where \( k \) is the size of the underlying metric space known before the execution and \( \Delta \) is the aspect ratio. Later, Azar et al. [4] proposed an almost-deterministic algorithm with competitive ratio \( O(\log k) \). Ashlagi et al. [2] analyzed Emek et al.’s algorithm in a simplified way, and improved its competitive ratio to \( O(\log k) \). They also extended these algorithms to bipartite matching with delays (MBPMD). The best known lower bound for MPMD is \( \Omega(\log k / \log \log k) \) and MBPMD \( \Omega(\sqrt{\log k / \log \log k}) \) [2]. In contrast to our work, all these papers assume that the time cost of a request is linear in its waiting time.

In contrast to this previous work, we focus on the uniform metric, i.e., the distance between any two points is the same. While this is only a special case, it is an important one.

In the existing linear-MPMD algorithms, a common step is to first embed a general metric to a probabilistic hierarchical separated tree (HST), which is actually an offline approach, and then design an online algorithm on the HST metric. The online algorithms on HST metrics are essentially based on algorithms on uniform metrics (or aspect-ratio-bounded metrics which can also be handled by our results) because every level of an HST can be considered as a uniform metric. Uniform metrics are known to be tricky, e.g., Emek et al. [11] study linear-MPMD with only two points. Uniform metrics also play an important role in the field of online computation [14]. For example, the \( k \)-server problem restricted to uniform metrics is the well-known paging problem.
The idea of delaying decisions has been around for a long time in the form of rent-or-buy problems (most prominently: ski rental), but [10] showed how to use delays in the context of combinatorial problems such as matching. In the classical ski rental problem [14], one can also consider the variation that the renting cost rate (to simplify our discussion, let’s consider the continuous case) may change over time. If the purchase price is a constant, the renting cost rate function does not change the competitive ratio since a good deterministic online algorithm is always to buy it when the renting fee is equal to the purchase price.

Azar et al. [5] considered online service with delay, which generalizes the k-server problem. As mentioned in their paper, delay penalty functions are not restricted to be linear and even different requests can have different penalty functions. However, different delay penalty functions there do not make the service with delay problem much different, and there is a universal way to deal with these different penalty functions, unlike the online matching problems we consider now.

3 Preliminaries

In this section, we formulate the problem and introduce notations.

3.1 Problem Statement

Let \( \mathbb{R}^+ \) stands for the set of nonnegative real numbers.

A metric space \( S = (V, \mu) \) is a set \( V \), whose members are called points, equipped with a distance function \( \mu : V^2 \rightarrow \mathbb{R}^+ \) which satisfies the following conditions

- **Positive definite**: \( \mu(x, y) \geq 0 \) for any \( x, y \in V \), and “=” holds if and only if \( x = y \);
- **Symmetric**: \( \mu(x, y) = \mu(y, x) \) for any \( x, y \in V \);
- **Subadditive or triangle inequality**: \( \mu(x, y) + \mu(y, z) \geq \mu(x, z) \) for any \( x, y, z \in V \).

Given a function \( f : \mathbb{R}^+ \rightarrow \mathbb{R}^+ \), the problem \( f \)-MPMD is defined as follows, and \( f \) is called the time cost function.

For any finite metric space \( S = (V, \mu) \), an online input instance over \( S \) is a set \( R \) of requests, with any \( \rho \in R \) characterized by its location \( t(\rho) \in V \) and arrival time \( t(\rho) \in \mathbb{R}^+ \). Each request \( \rho \) is revealed exactly at time \( t(\rho) \). Assume that \( |R| \) is an even number. The goal is to construct a perfect matching, i.e. a partition into pairs, of the requests in real time without preemption.

Suppose an algorithm \( A \) matches \( \rho, \rho' \in R \) at time \( T \). It pays the space cost \( \mu(t(\rho), t(\rho')) \) and the time cost \( f(T - t(\rho)) + f(T - t(\rho')) \). The space cost of \( A \) on input \( R \), denoted by \( \text{cost}_{A}(R) \), is the total space cost caused by all the matched pairs, and the time cost \( \text{cost}_{A}(R) \) is defined likewise. The objective of the \( f \)-MPMD is to find an online algorithm \( A \) such that \( \text{cost}_{A}(R) = \text{cost}_{A}(R) + \text{cost}_{A}'(R) \) is minimized for all \( R \).

As usual, the online algorithm \( A \) is evaluated through competitive analysis. Let \( A^* \) be an optimum offline algorithm\(^2\). For any finite metric space \( S \), if there are \( a, b \in \mathbb{R}^+ \) such that \( \text{cost}_{A}(R) \leq \text{cost}_{A^*}(R) + a + b \) for any online input instance \( R \) over \( S \), then \( A \) is said to be \( a \)-competitive on \( S \). The minimum such \( a \) is called the competitive ratio of \( A \) on \( S \). Note that both \( a \) and \( b \) can depend on \( S \).

This paper will focus on monomial time cost functions \( f(t) = t^\alpha, \alpha > 1 \) and uniform metric spaces. A metric space \( (V, \mu) \) is called \( \delta \)-uniform if \( \mu(u, v) = \delta \) for any \( u, v \in V \).

\(^2\) An offline algorithm knows the whole input instance at the beginning and outputs any pair \( \rho, \rho' \in R \) at time \( \max\{t(\rho), t(\rho')\} \).
A natural idea to solve the $f$-MPMD problem is to prioritize internal matches and to create an external match only if both requests have waited long enough (say, as long as $\theta$). However, for any monomial time cost function $f(t) = t^\alpha, \alpha > 1$, the strategy (called Strategy I) is not competitive, as illustrated in Example 4.

**Example 4.** For any positive integer $n$ and small real number $\epsilon > 0$, construct an online instance as follows. A request $p_i$ arrives at time $i \cdot \theta$ for any $0 \leq i \leq n$, while a request $p_{i-1}$ arrives at time $(i-1) \cdot \theta - \epsilon$ for any $1 \leq i \leq n$. Point $v$ gets a request $p'_i$ at time 0. By Strategy I, as in Figure 1(a), each $p_i$ is matched with $p_{i+1}$ for any $0 \leq i < n$, and $p'_i$ and $p_{2i}$ are matched, causing cost at least $n \cdot f(\theta - \epsilon) + f(n \theta) + \delta$. Consider the offline solution consisting of $\langle p'_i, p_0 \rangle$ and $\langle p_{2i-1}, p_{2i} \rangle$ for $1 \leq i \leq n$, as in Figure 1(b), which has cost $\delta + n \cdot f(\epsilon)$. When $n$ approaches infinity and $\epsilon$ approaches 0, $n \cdot f(\theta - \epsilon) + f(n \theta) + \delta \gg \delta + n \cdot f(\epsilon)$, meaning that Strategy I is not competitive.

A plausible way to improve Strategy I is to accumulate the time costs of all the co-located requests which arrive after the last external match involving the point, and to enable an external match if both points have accumulated enough costs (say, as large as $\theta$). Though applicable to the scenario in Example 4, this improvement (called Strategy II) remains not competitive for any time cost function $f(t) = t^\alpha, \alpha > 1$, as shown in the next example.

**Example 5.** Again, consider two points $u, v$ of distance $\delta$. Arbitrarily fix an even integer $n > 0$ and a small real number $\epsilon > 0$. Arbitrarily choose $\tau \in \mathbb{R}^+$ such that $\theta - \epsilon < \frac{1}{2} f(\tau) + \delta$. Suppose that a request $p'_i$ arrives at $v$ at time 0, while a request $p_i$ arrives at $u$ at time $i \tau$ for any $0 \leq i \leq n$. Hence there are totally $n + 2$ requests. As illustrated in Figure 2(a), applying Strategy II results in the matches $\langle p'_i, p_0 \rangle$ and $\langle p_i, p_{i+1} \rangle$ for any even number $0 \leq i < n$, causing cost at least $\frac{1}{2} f(\tau) + f(n \tau) + \delta$. On the other hand, consider the offline solution $\langle p'_i, p_0 \rangle$ and $\langle p_i, p_{i+1} \rangle$ for any odd number $0 < i < n$, as shown in Figure 2(b). It has cost $\delta$.

3.2 Notations and Terminologies

Any pair of requests $\rho, \rho'$ in the perfect matching is called a match between $\rho$ and $\rho'$ and denoted by $\langle \rho, \rho' \rangle$ or $\langle \rho', \rho \rangle$ interchangeably. A match $\langle \rho, \rho' \rangle$ is said to be external if $f(\rho) \neq f(\rho')$, and internal otherwise. For any request $\rho$, let $T(\rho)$ be the time when $\rho$ is matched; $\rho$ is said to be pending at any time $t \in (t(\rho), T(\rho))$ and active at any time $t \in [t(\rho), T(\rho)]$. At any moment $t$, a point $v \in V$ is called aligned if the number of pending requests at $v$ under $A$ and that under $A^*$ have the same parity, and misaligned otherwise. The derivative of any differentiable function $f : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ is denoted by $f'$.

4 Algorithm and Analysis

4.1 Basic Ideas

A natural idea to solve $f$-MPMD on uniform metrics is to prioritize internal matches and to create an external match only if both requests have waited long enough (say, as long as $\theta$). However, for any monomial time cost function $f(t) = t^\alpha, \alpha > 1$, the strategy (called Strategy I) is not competitive, as illustrated in Example 4.

**Example 4.** For any positive integer $n$ and small real number $\epsilon > 0$, construct an online instance as follows. A request $p_i$ arrives at $u$ at time $i \cdot \theta$ for any $0 \leq i \leq n$, while a request $p_{i-1}$ arrives at $u$ at time $(i-1) \cdot \theta - \epsilon$ for any $1 \leq i \leq n$. Point $v$ gets a request $p'_i$ at time 0. By Strategy I, as in Figure 1(a), each $p_i$ is matched with $p_{i+1}$ for any $0 \leq i < n$, and $p'_i$ and $p_{2i}$ are matched, causing cost at least $n \cdot f(\theta - \epsilon) + f(n \theta) + \delta$. Consider the offline solution consisting of $\langle p'_i, p_0 \rangle$ and $\langle p_{2i-1}, p_{2i} \rangle$ for $1 \leq i \leq n$, as in Figure 1(b), which has cost $\delta + n \cdot f(\epsilon)$. When $n$ approaches infinity and $\epsilon$ approaches 0, $n \cdot f(\theta - \epsilon) + f(n \theta) + \delta \gg \delta + n \cdot f(\epsilon)$, meaning that Strategy I is not competitive.

A plausible way to improve Strategy I is to accumulate the time costs of all the co-located requests which arrive after the last external match involving the point, and to enable an external match if both points have accumulated enough costs (say, as large as $\theta$). Though applicable to the scenario in Example 4, this improvement (called Strategy II) remains not competitive for any time cost function $f(t) = t^\alpha, \alpha > 1$, as shown in the next example.

**Example 5.** Again, consider two points $u, v$ of distance $\delta$. Arbitrarily fix an even integer $n > 0$ and a small real number $\epsilon > 0$. Arbitrarily choose $\tau \in \mathbb{R}^+$ such that $\theta - \epsilon < \frac{1}{2} f(\tau) + \delta$. Suppose that a request $p'_i$ arrives at $v$ at time 0, while a request $p_i$ arrives at $u$ at time $i \tau$ for any $0 \leq i \leq n$. Hence there are totally $n + 2$ requests. As illustrated in Figure 2(a), applying Strategy II results in the matches $\langle p'_i, p_0 \rangle$ and $\langle p_i, p_{i+1} \rangle$ for any even number $0 \leq i < n$, causing cost at least $\frac{1}{2} f(\tau) + f(n \tau) + \delta$. On the other hand, consider the offline solution $\langle p'_i, p_0 \rangle$ and $\langle p_i, p_{i+1} \rangle$ for any odd number $0 < i < n$, as shown in Figure 2(b). It has cost $\delta$. 

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example_diagram.png}
\caption{The input instance of Example 4. A blue dot stands for a request, and a thick line or curve for a match. (a) is the matching produced by Strategy I, while (b) is an offline solution.}
\end{figure}
Impatient Online Matching

Figure 2 The input instance of Example 5. A blue dot stands for a request, and a thick line or curve for a match. (a) is the matching produced by Strategy II, while (b) is an offline solution.

Figure 3 The input instance of Example 6. A blue dot stands for a request, an area surrounded by dash lines stands for a part of the instance, and a thick line or curve for a match. (a) is the matching produced by Strategy III, while (b) is an offline solution.

\[ \frac{n}{2} f(\tau) + \delta. \] Thus the cost of \( A^* \) is at most \( \frac{n}{2} f(\tau) + \delta \). When \( n \) approaches infinity and \( \epsilon \) approaches 0, we have \( \frac{n}{2} f(\tau) + f(n \tau) + \delta \gg \frac{n}{2} f(\tau) + \delta \), implying that Strategy II is not competitive.

Since the trouble may be rooted at the double-counter-enabling mechanism which enables an external match when two counters both reach some threshold, we further improve the strategy by enabling an external match if one of the two points has high accumulated cost (say, as high as \( \theta \)). This improvement (called Strategy III) defeats both Examples 4 and 5, but the following example shows that it remains not competitive for any monomial time cost function \( f(t) = t^\alpha, \alpha > 1 \).

Example 6. Choose \( \tau \in \mathbb{R}^+ \) and odd integer \( n > 0 \) such that \( f(n \tau) = \theta \). Arbitrarily choose real number \( T_0 > f^{-1}(\theta) \). Consider a uniform metric space \( S = (\{u, v, w\}, \delta) \). Let \( m > 0 \) be an arbitrary integer. Construct an online input instance \( R \) which is the union of \( m+1 \) parts \( R_0, \ldots, R_m \), as illustrated in Figure 3.

The part \( R_0 \) has \( 5n + 3 \) requests. Specifically, \( u \) receives a request \( \rho^u_{0,0} \) at time 0, \( \rho^u_{0,i} \) at time \( T_0 + (n + i) \tau \) for any \( 1 \leq i \leq 2n \). \( v \) receives a request \( \rho^v_{0,i} \) at time \( T_0 + i \tau \) for any \( 1 \leq i \leq 2n \). \( w \) receives a request \( \rho^w_{0,i} \) at time 0 and a request \( \rho^w_{0,n+i} \) at time \( T_0 + i \tau \) for any \( 1 \leq i \leq n \). Let \( T_1 = T_0 + (2n + 1) \tau, T_j = T_{j-1} + 3n \tau \) for any \( 2 \leq j \leq m \).

For any \( 1 \leq j \leq m \), the part \( R_j \) has \( 6n \) requests as follows: \( \rho^u_{j,i} \) arrives at \( u \) at time \( T_j + (2n + i - 1) \tau \), \( \rho^v_{j,i} \) arrives at \( v \) at time \( T_j + (n + i - 1) \tau \), and \( \rho^w_{j,i} \) arrives at \( w \) at time \( T_j + (i - 1) \tau \), for every \( 1 \leq i \leq 2n \).
Actually, we can very slightly perturb the arrival time of some requests so that Strategy III results in exactly the following external matches: $\langle \rho_{0,-m}, \rho_{0,-m+1} \rangle$, $\langle \rho_{0,0}, \rho_{0,n} \rangle$, $\langle \rho_{n,2n}, \rho_{2n+2n} \rangle$ for $1 \leq j \leq m$, $\langle \rho_{n,2n}, \rho_{n+1,2n} \rangle$ and $\langle \rho_{n,2n}, \rho_{n+1,2n} \rangle$ for $1 \leq i < m$, and $\langle \rho_{m,2n}, \rho_{m,2n} \rangle$, as illustrated in Figure 3(a). The cost of Strategy III is at least $3m(\delta + \theta)$. On the other hand, consider the offline solution SOL which has no external matches, as indicated in Figure 3(b). It has cost at most $2f(T_0 + \tau) + \frac{6mn+5n-1}{2}f(\tau)$. When $\tau$ approaches zero and $m$ approaches infinity, we have $3m(\delta + \theta) \gg 2f(T_0 + \tau) + \frac{6mn+5n-1}{2}f(\tau)$, implying that Strategy III is not competitive.

Let’s look closer at the example. Consider an arbitrary (except the first) external match $\langle \rho, \rho' \rangle$ of Strategy III. It is of misaligned-aligned pattern in the sense that $\ell(\rho)$ and $\ell(\rho')$ have opposite alignment status when the match occurs. Suppose $\ell(\rho)$ is misaligned. Then it has accumulated high cost, mainly due to the long delay of $\rho$. On the contrary, SOL has accumulated little cost at $\ell(\rho)$, because SOL has no pending request there while $\rho$ is pending. Hence, a match of misaligned-aligned pattern can significantly enlarge the gap between online/offline costs. To be worse, such a match does not change the number of aligned/misaligned points, making it possible that this pattern appears again and again, enlarging the gap infinitely. As a result, we establish a set which consists of points that are likely to be misaligned, and prioritize matching those requests that are located outside the set. The algorithm is described in detail as follows.

4.2 Algorithm Description

Our algorithm maintains a subset $\Psi \subseteq V$ and a counter $z_v \in \mathbb{R}^+$, which is initially set to 0, for every point $v \in V$. The algorithm proceeds round by round, and $\Psi$ is reset to be the empty set $\emptyset$ at the beginning of each round. The first round begins when the algorithm starts. Let $k = |V|$. Whenever $2k$ external matches are output, the present round ends immediately and the next one begins. At any time $t$, the following operations are performed exhaustively, i.e., until there is no possible matching according to the following rules.

1. Every $z_v$ increases at rate $f'(t - t_0)$ if there is an active request $\rho$ at $v$ with $t(\rho) = t_0$.
2. Match any pair of active requests $\rho$ and $\rho'$ if $\ell(\rho) = \ell(\rho')$.
3. For any pair of active requests $\rho, \rho'$ with $u \triangleq \ell(\rho) \neq v \triangleq \ell(\rho')$, match them and reset $z_u = z_v = 0$ if there is $x \in \{u, v\}$ satisfying
   a. $z_x \geq 2\delta$, or
   b. $\delta \leq z_x < 2\delta$ and $(\{u, v\} \cap \Psi = \emptyset$.

   Arbitrarily choose such an $x \in \{u, v\}$, and we say that $x$ initiates this match. Reset $\Psi$ to be $(\Psi \setminus \{u, v\}) \cup \{x\}$ if either $u \notin \Psi$ or $v \notin \Psi$. Priority rule: in applying Operation 3, the requests located outside $\Psi$ are prioritized.

4.3 Competitive Analysis

Throughout this subsection, arbitrarily fix a time cost function $f(t) = t^\alpha$ with $\alpha > 1$, a uniform metric space $\mathcal{S} = (V, \delta)$ of $k$ points, and an arbitrary online input instance $R$ over $\mathcal{S}$. For ease of presentation, we assume that the arrival times of the requests are pairwise different. This assumption does not lose generality since the arrival times can be arbitrarily perturbed and timing in practice is up to errors. Let $\mathcal{A}$ stand for our algorithm and $\mathcal{A}^*$ for an optimum offline algorithm solving $f$-MPMD. We start competitive analysis by introducing notation.
4.3.1 Notations

For any request $\rho \in R$ and subset $I \subseteq \mathbb{R}^+$ of time, the time cost of $A^*$ incurred by $\rho$ during $I$ is defined to be

$$C_{\text{time}}(\rho, I, A^*) = \int_{(t(\rho), T^*(\rho]) \cap I} f'(t - t(\rho)) dt,$$

where $T^*(\rho)$ is the time when $\rho$ gets matched by $A^*$. For any $v \in V$, define

$$C_{\text{time}}(v, I, A^*) = \sum_{\rho \in R, t(\rho) = v} C_{rime}(\rho, I, A^*).$$

Let $C_{\text{space}}(v, I, A^*)$ be $\frac{1}{2}$ times the number of requests at $v$ that are externally matched by $A^*$ during $I$.

Define $\Gamma = \{t \in \mathbb{R}^+ : \text{at time } t, A \text{ has a pending request } \rho \text{ with } z_t(\rho) > 2\delta\}$. We will analyze time cost of $A^*$ inside and outside $\Gamma$ separately.

Our algorithm $A$ runs round by round. Specifically, the round starting at time $t_0$ and ending at time $t_1$ is referred to as the time period $(t_0, t_1]$. Let $\Pi$ be the set of rounds of $A$.

For any $\pi \in \Pi$, define $\text{round\_cost}_{\text{time}}(\pi, A^*) = \sum_{v \in V} C_{\text{time}}(v, \pi \setminus \Gamma, A^*)$ which stands for the time cost of $A^*$ during $\pi \setminus \Gamma$, and $\text{round\_cost}_{\text{space}}(\pi, A^*) = \sum_{v \in V} C_{\text{space}}(v, \pi, A^*)$ which is the space cost of $A^*$ during $\pi$.

For any $v \in V$, we divide time into phases based on $A$’s behavior as follows. The first phase begins at time $t = 0$. Whenever an external match involving $v$ occurs, the current phase of $v$ ends and the next phase of $v$ begins. Specifically, the phase of $v$ starting at time $t_0$ and ending at time $t_1$ is referred to as the period $(t_0, t_1]$ spent by $v$. For any $v \in V$, let $\Phi_v$ be the set of phases of $v$, and $\Phi = \bigcup_{v \in V} \Phi_v$. For any $\phi \in \Phi_v$, define the value of $\phi$, denoted by $\sigma(\phi)$, to be the value of $z_v$ at the end of $\phi$. For an external match $m$ of $A$ initiated by $v$, the phase of $v$ ending with $m$ is called the phase of $m$, denoted by $\phi_m$. For any round $\pi \in \Pi$, let $\Phi_\pi$ be the set of phases ending in $\pi$. For any round $\pi \in \Pi$, define $\text{phase\_cost}_{\text{time}}(\pi, A^*) = \sum_{v \in V} \sum_{\phi \in \Phi_\pi} \bigcap_{\phi_v} C_{\text{time}}(v, \phi \setminus \Gamma, A^*)$, and $\text{phase\_cost}_{\text{space}}(\pi, A^*) = \sum_{v \in V} \sum_{\phi \in \Phi_\pi} \bigcap_{\phi_v} C_{\text{space}}(v, \phi, A^*)$.

We say that a phase of $v$ is good, if the alignment status of $v$ does not change during the phase. Furthermore, a round $\pi$ is good if all the phases in $\Phi_\pi$ are good. A phase or a round is said to be bad if it is not good.

A phase is called complete if it ends with an external match of $A$, while a round is complete if $A$ outputs $2k$ external matches during it. Obviously, any round other than the final one is complete.

4.3.2 Competitive Ratio of Our Algorithm

Basically, we show that in every round, the incremental cost of $A$ and that of $A^*$ do not differ too much. This is reduced to two tasks. First, if all the counters are always small (say, no more than $4\delta$), the incremental cost of $A$ in every round is $O(kd)$, so it suffices to show that the cost of $A^*$ increases by $\Omega(d)$. This is the main task of this subsection and presented in Lemma 8. Second, to deal with the case that some counter $z_v$ is large, we have to show that the accumulated cost of $A^*$ in the phase increases nearly proportionately with $z_v$, as claimed in Lemma 9.

The following is a key lemma, stating that in every good complete round of $A$, the cost of the optimum offline algorithm $A^*$ is not small.
Lemma 7. In every good complete round \( \pi \), we have either \( \text{round\_cost\_time}(\pi, A^*) \geq f(f^{-1}(2\delta) - f^{-1}(\delta)) \) or \( \text{round\_cost\_space}(\pi, A^*) \geq \delta \), or \( \text{phase\_cost\_time}(\pi, A^*) \geq \delta \).

Up to now, we have focused on good rounds. The next lemma indicates that the cost of \( A^* \) in bad rounds can be ignored in some sense.

Lemma 8. The number of bad rounds of \( A \) is at most twice the number of external matches of \( A^* \).

For any phase \( \phi \in \Phi \), define its truncated value to be
\[
\sigma'(\phi) = \begin{cases} 
0 & \text{if } \sigma(\phi) \leq 2\delta, \\
f(f^{-1}(\sigma(\phi)) - f^{-1}(2\delta)) & \text{otherwise}.
\end{cases}
\]

We will use truncated phase values to give a lower bound of the time cost of \( A^* \).

Lemma 9. \( \text{cost}_{A^*}(R) \geq \sum_{\pi \in \Pi} \text{phase\_cost\_time}(\pi, A^*) + \sum_{\phi \in \Phi} \sigma'(\phi) \).

The following technical lemmas will be needed.

Lemma 10. For any \( c_1, \ldots, c_n \geq c_0 > c > 0 \) and \( \alpha > 1 \), we have
\[
\frac{\sum_{j=1}^{n}(c_j - c)}{\sum_{j=1}^{n}(\sqrt[\alpha]{c_j} - \sqrt[\alpha]{c})^\alpha \leq \frac{c_0 - c}{(\sqrt[\alpha]{c_0} - \sqrt[\alpha]{c})^\alpha}.
\]

Lemma 11. If \( A \) has only one round on the instance \( R \), \( \text{cost}_{A}(R) / \text{cost}_{A^*}(R) = O(k) \).

Now we are ready to prove the main result.

Theorem 1. For any \( f(t) = t^\alpha \) with constant \( \alpha > 1 \), the competitive ratio of \( A \) for \( f\text{-MPMD} \) on \( k \)-point uniform metric space is \( O(k) \).

Proof. Suppose that \( A \) has \( m \) rounds on the online input instance \( R \), namely \( |\Pi| = m \). By Lemma 11, we assume that \( m > 1 \).

In every round, there are at most \( 2k \) external matches and each of them ends two complete phases. So, there are altogether at most \( 4km \) complete phases. Considering that there are totally at most \( k \) incomplete phases, \( |\Phi| \leq (4m+1)k \leq 5mk \). Let \( \Phi' = \{ \phi \in \Phi : \sigma(\phi) \geq 4\delta \} \). It holds that \( \text{cost}_{A^*}(R) = \text{cost}_{A^*}(R) + \text{cost}_{A^*}(R) \leq 2k\delta + \sum_{\phi \in \Phi} \sigma(\phi) \leq 22k\delta + \sum_{\phi \in \Phi} (\sigma(\phi) - 4\delta) \leq 22k\delta + \sum_{\phi \in \Phi} (\sigma(\phi) - 2\delta) \).

On the other hand, as to the cost of \( A^* \), we have \( \text{cost}_{A^*}(R) = \text{cost}_{A^*}(R) + \text{cost}_{A^*}(R) \geq \sum_{\pi \in \Pi} \text{phase\_cost\_time}(\pi, A^*) + \sum_{\phi \in \Phi} \sigma'(\phi) \) by Lemma 9. Trivially we also have \( \text{cost}_{A^*}(R) \geq \sum_{\pi \in \Pi} \text{round\_cost\_time}(\pi, A^*) + \text{round\_cost\_space}(\pi, A^*) \). Let \( \Pi' \) be the set of good complete rounds and \( m' = |\Pi'| \). Let \( m'' \) be the number of bad rounds. An easy observation is that \( m' + m'' \geq m - 1 \). By Lemma 8, \( A^* \) has at least \( m''\alpha \) external matches:

\[
2\text{cost}_{A^*}(R) \geq \text{cost}_{A^*}(R) + \sum_{\pi \in \Pi} \text{phase\_cost\_time}(\pi, A^*) + \sum_{\phi \in \Phi} \sigma'(\phi) \\
+ \sum_{\pi \in \Pi}[\text{round\_cost\_time}(\pi, A^*) + \text{round\_cost\_space}(\pi, A^*)] \\
\geq \frac{m''}{\alpha} \delta + \sum_{\phi \in \Phi} \sigma'(\phi) + \sum_{\pi \in \Pi} \text{phase\_cost\_time}(\pi, A^*) \\
+ \text{round\_cost\_time}(\pi, A^*) + \text{round\_cost\_space}(\pi, A^*)] \\
\geq \frac{m''}{\alpha} \delta + \sum_{\phi \in \Phi} \sigma'(\phi) + f(f^{-1}(\delta) - f^{-1}(2\delta))m'' \\
\geq \frac{m''}{\alpha} (\sqrt[\alpha]{2} - 1)^\alpha \delta + \sum_{\phi \in \Phi} \sigma'(\phi)
\]

where the third equality is due to Lemma 7.

Altogether, \( \frac{\text{cost}_{A}(R)}{\text{cost}_{A^*}(R)} \leq \frac{22k\delta + \sum_{\phi \in \Phi} (\sigma(\phi) - 2\delta)}{m''(\sqrt[\alpha]{2} - 1)^\alpha \delta + \sum_{\phi \in \Phi} \sigma'(\phi)} \), which is \( O(k) \) by Lemma 10. \( \square \)
5 Lower Bound for Deterministic Algorithms

This section is devoted to showing that any deterministic algorithm for the convex-MPMD problem on \(k\)-point uniform metric space must have competitive ratio \(\Omega(k)\), meaning that our algorithm is optimum, up to a constant factor. Let's begin with a convention of notation. Let \(f : \mathbb{R}^+ \mapsto \mathbb{R}^+\) be a nondecreasing, unbounded, continuous function satisfying \(f(0) = f'(0) = 0\). Let \(S = (V, \delta)\) be a uniform metric space with \(V = \{v_0, v_1, \ldots, v_k\}\). Suppose that \(A\) is an arbitrary deterministic online algorithm for the \(f\)-MPMD problem. Let \(T \in \mathbb{R}^+\) be such that \(f(T) = k\delta\). Arbitrarily choose a real number \(\tau > 0\) such that \(n = \frac{T}{\tau}\) is an even number.

We construct an instance \(R\) of online input to \(A\) and show that the competitive ratio of \(A\) is at least \(\Omega(k)\). The instance \(R\) is determined in an online fashion: Roughly speaking, based on the up-to-now behavior of \(A\), we choose when and where to input next requests so as to force \(A\) to have many external matches.

Specifically, \(R\) is determined in \(m\) rounds, where \(m\) is an arbitrary positive integer. The first round begins at time \(T_1 = 0\). Some requests arrive in the manner as described in the next four paragraphs. At arbitrary time \(T_2\) after these requests are all matched, finish the first round and start the second round. Repeat this process until we have finished \(m\) rounds.

All the requests form the instance \(R\).

Now we describe the requests that arrive during the \(r\)th round, namely in the interval \([T_r, T_{r+1})\), for any \(1 \leq r \leq m\). Basically, at \(v_0\) there is just one request, denoted by \(\rho_{00}\), which arrives at time \(T_r\), while a request \(\rho_{ij}\) arrives at every point \(v_i\) at time \(T_r + j\tau\), for any integers \(1 \leq i \leq k\) and \(j \geq 1\). We will iteratively specify when requests should stop arriving at the points other than \(v_0\).

Define \(G_0 = (V, \emptyset)\) to be the graph on \(V\) with no edges. Let \(C_0 = \{v_0\}\). Starting with \(h = 1\), iterate the following process until no more requests will arrive. At time \(T_r + hT\), construct an undirected graph \(G_h\) on \(V\). It has an edge between any pair of vertices \(v_i \neq v_j\) if and only if by time \(T_r + hT\), \(A\) has matched one request at \(v_i\) and another at \(v_j\) both of which arrived during the period \([T_r, T_r + hT]\). Let \(C_h\) be the set of the vertices in the connected component of \(G_h\) containing \(v_0\). We proceed case by case:

**Case 1:** \(C_{h-1} \neq C_h\)\(V\). Then no more requests except \(\rho_{i,hn+1}\) will arrive, where \(i\) is arbitrarily chosen such that \(v_i \in C_h \setminus C_{h-1}\). Denote this \(h\) by \(h_r\).

**Case 2:** \(C_{h-1} = C_h\). Then no more requests except \(\rho_{i,hn+1}\) will arrive, where \(i\) is arbitrarily chosen such that \(v_i \in V \setminus C_h\). Denote this \(h\) by \(h_r\).

**Case 3:** otherwise. Then no more requests will arrive at any \(v_i \in C_h\), while requests continue arriving at points in \(V \setminus C_h\). Increase \(h\) by 1 and iterate.

Arbitrarily fix \(1 \leq r \leq m\) in the rest of this section. Let \(R_r\) be the set of requests that arrive in the first \(r\) rounds, and \(N_r\) be the number of requests in \(R_r \setminus R_{r-1}\), where \(R_0 = \emptyset\).

Let \(R = R_m\). It is easy to see four facts:

**Fact 1:** \(N_r \leq k^2 n + 2\).

**Fact 2:** \(R_r \setminus R_{r-1}\) has exactly one request at \(v_0\), and has an odd number of requests at the point where the last request arrives, respectively.

**Fact 3:** \(R_r \setminus R_{r-1}\) has an even number of requests at any other point.

**Fact 4:** No match occurs between requests of different rounds.

Some lemmas are needed for proving the main result.

- **Lemma 12.** \(\text{cost}_A(R_r) \leq (\delta + \frac{k^2 n}{2} f(\tau) + f(\tau)) r\).

- **Lemma 13.** \(\text{cost}_A(R_r) \geq k\delta r\).
Theorem 3. Suppose that the time cost function \( f \) is nondecreasing, unbounded, continuous and satisfies \( f(0) = f'(0) = 0 \). Then any deterministic algorithm for \( f \)-MPMD on \( k \)-point uniform metric space has competitive ratio \( \Omega(k) \).

Proof. Suppose there are \( a = a(k, \delta) \) and \( b = b(k, \delta) \) such that for any \( m \geq 1 \), \( \text{cost}_A(R) \leq a \cdot \text{cost}_A(R) + b \). Fix \( k \) and \( \delta \). Dividing both sides of inequality by \( m \) and letting \( m \) approach infinity, by Lemmas 12 and 13, we get \( f(n\tau) \leq (\delta + \frac{k\delta}{2}f(\tau) + f(\tau))a \), which means that

\[
a \geq \frac{f(n\tau)}{\delta + \frac{k\delta}{2}f(\tau) + f(\tau)} = \frac{\frac{f(n\tau)}{\delta} + \frac{1}{2}f(n\tau)}{\delta + \frac{k\delta}{2}f(\tau) + f(\tau)}.
\]

Let \( \tau \) approach zero. One has \( \lim_{\tau \to 0} f(\tau) = 0 \), and

\[
\lim_{\tau \to 0} \frac{f(n\tau)}{\tau^2} = \lim_{\tau \to 0} \frac{f(n\tau)}{\tau} \cdot \frac{\tau}{f(\tau)} = \lim_{\tau \to 0} \frac{1}{\tau^2} \cdot \frac{f(T)}{f(\tau)} \cdot \frac{\tau}{f(\tau)} = +\infty \quad \text{since } f'(0) = 0
\]

This means \( \lim_{\tau \to 0} k^2 n f(\tau) = 0 \), since \( f(n\tau) = k\delta \) is a constant when \( k \) and \( \delta \) are fixed. As a result, \( a = \lim_{\tau \to 0} a \geq \lim_{\tau \to 0} \frac{\frac{f(n\tau)}{\delta} + \frac{1}{2}f(n\tau)}{\delta + \frac{k\delta}{2}f(\tau) + f(\tau)} = \frac{k\delta}{\delta} = k \).

6 Conclusion

We have designed an optimum deterministic online algorithm that solves \( f \)-MPMD for any mononial function \( f(t) = t^\alpha \) with \( \alpha > 1 \). It is remarkable that the algorithm remains optimum if only \( f : \mathbb{R}^+ \to \mathbb{R}^+ \) is an increasing and convex polynomial function with \( f(0) = 0 \). Actually, following Subsection 4.3.2, one can easily see that the competitive ratio is at most

\[
\max \left\{ \frac{1}{f^{-1}(\frac{28}{45})} \sup_{c \geq 28} \left\{ \frac{c-28}{f^{-1}(c) - f^{-1}(28)} \right\} \right\},
\]

which is \( O(k) \) by elementary calculus, when \( f \) is fixed.

An interesting future direction is to design a randomized algorithm for convex-MPMD. A randomized algorithm is usually more competitive than a deterministic one when considering oblivious adversaries. We conjecture that there is a randomized algorithm for convex-MPMD with competitive ratio \( O(\log k) \) but no such algorithm with competitive ratio \( O(1) \). If this turns out true, there is still a clear separation between linear-MPMD and convex-MPMD in the context of randomized algorithms.

In contrast to convex functions, concave functions may model the fact that in some applications the delay cost grows slower and slower, which encourages matching two new requests instead of matching old requests. It seems not difficult to design an algorithm with bounded competitive ratio for these concave cost functions, but to design a good one, i.e., with a very small competitive ratio, seems still challenging.

References


Impatient Online Matching

Extensions of Self-Improving Sorters

Siu-Wing Cheng
HKUST, Hong Kong, China
scheng@cse.ust.hk
https://orcid.org/0000-0002-3557-9935

Lie Yan
Hangzhou, China
lieyan@yahoo.com

Abstract

Ailon et al. (SICOMP 2011) proposed a self-improving sorter that tunes its performance to the unknown input distribution in a training phase. The distribution of the input numbers \(x_1, x_2, \ldots, x_n\) must be of the product type, that is, each \(x_i\) is drawn independently from an arbitrary distribution \(D_i\), and the \(D_i\)'s are independent of each other. We study two extensions that relax this requirement. The first extension models hidden classes in the input. We consider the case that numbers in the same class are governed by linear functions of the same hidden random parameter. The second extension considers a hidden mixture of product distributions.

2012 ACM Subject Classification
Theory of computation → Design and analysis of algorithms

Keywords and phrases
sorting, self-improving algorithms, entropy

Digital Object Identifier
10.4230/LIPIcs.ISAAC.2018.63

1 Introduction

Self-improving algorithms proposed by Ailon et al. [1] can tune their computational performance to the input distribution. There is a training phase in which the algorithm learns certain input features and computes some auxiliary structures. After the training phase, the algorithm uses these auxiliary structures in the operation phase to obtain an expected time complexity that is no worse and possibly better than the best worst-case complexity known. The expected time complexity in the operation phase is called the limiting complexity.

This computational model addresses two issues. First, the worst-case scenario may not happen, and so the worst-case optimal performance may not be the best possible. Second, previous efforts for mitigating the worst-case scenarios often consider average-case complexities, and the input distributions are assumed to be simple distributions like Gaussian, uniform, Poisson, etc. whose parameters are given beforehand. In contrast, Ailon et al. only assume that individual input items are independently distributed, while the distribution of an input item can be arbitrary. No other information is needed.

The problems of sorting and two-dimensional Delaunay triangulation are studied by Ailon et al. [1]. The sorting problem input \(I\) has \(n\) numbers. The \(i\)-th number is drawn from a hidden distribution \(D_i\), and the \(D_i\)'s are independent from each other. The joint distribution \(\prod_{i=1}^{n} D_i\) is called a product distribution. Let \(\pi(I)\) denote the sequence of the ranks of the \(x_i\)'s, which is a permutation of \([n]\). It is shown that for any \(\varepsilon \in (0, 1)\), there is a self-improving algorithm with limiting complexity \(O(\varepsilon^{-1}(n + H_\pi))\), where \(H_\pi\) is the

---

1 Supported by Research Grants Council, Hong Kong, China (project no. 16200317).
2 Part of the work was conducted while the author was at HKUST and supported by the Hong Kong PhD Fellowship.
Extensions of Self-Improving Sorters

Ailon et al. showed that there is a hidden partition of the input instances in \( O(n) \) time, and it succeeds with probability at least \( 1 - 1/n \), i.e., the probability of achieving the desired limiting complexity is at least \( 1 - 1/n \). For two-dimensional Delaunay triangulation, Ailon et al. also obtained an optimal limiting complexity for product distributions.

Subsequently, Clarkson et al. [2] developed self-improving algorithms for two-dimensional coordinatewise maxima and convex hull, assuming that the input comes from a product distribution. The limiting complexities for the maxima and the convex hull problems are \( O(\text{OptM} + n) \) and \( O(\text{OptC} + n \log \log n) \), where \( \text{OptM} \) and \( \text{OptC} \) are the expected depths of optimal linear decision trees for the maxima and convex hull problems, respectively.

On one hand, the product distribution requirement is very strong; on the other hand, Ailon et al. showed that \( \Omega(2^n \log n) \) bits of storage are necessary for optimal sorting if the \( n \) numbers are drawn from an arbitrary distribution. We study two extensions of the input model that are natural and yet possess enough structure for efficient self-improving algorithms to be designed.

The first extension models the situations in which some input elements depend on each other. We consider a hidden partition of the input \( I = (x_1, \ldots, x_n) \) into classes \( S_k \)’s such that all \( x_i \’s \) in a class \( S_k \) are distinct linear functions of the same hidden random parameter \( z_k \), and the distributions of the \( z_k \’s \) are arbitrary and independent of each other.\(^3\) We call this model a product distribution with hidden linear classes. Choose any \( \varepsilon \in (0, 1) \). Our self-improving sorter has an \( O(n/\varepsilon + H_\varepsilon/\varepsilon) \) limiting complexity, uses \( O(n^2) \) space, and requires a training phase that processes \( O(n^2) \) input instances in \( O(n^2 \log^3 n) \) time with a success probability at least \( 1 - 1/n \).

In the second extension, the distribution of \( I \) is a mixture \( \sum_{q=1}^k \lambda_q D_q \), where \( k \) and the \( \lambda_q \’s \) are hidden, and every \( D_q \) is a hidden product distribution of \( n \) real numbers. In other words, over a large collection of input instances, for all \( q \in [1, k] \), a fraction \( \lambda_q \) of them are expected to be drawn from \( D_q \). We assume that an upper bound \( m \geq k \) is given. We call this model a hidden mixture of product distributions. For any \( \varepsilon \in (0, 1) \), our sorter has an \( O(n \log \log (mn) + (n/\varepsilon) \log k + H_\varepsilon/\varepsilon) \) limiting complexity, uses \( O(mn) \) space, and requires a training phase that processes \( O(m \log m + \log n) + n^2 \) instances in \( O(mn \log m + \log n)^2 + m^2 n^{1+\varepsilon} \) time with a success probability at least \( 1 - 1/n \).

\section{Hidden linear classes}

There is a hidden partition of \([n]\) into classes. For every \( i \in [1, n] \), the distribution of \( x_i \) is degenerate if \( x_i \) is equal to a fixed value. Each such \( x_i \) will be recognized in the training phase and \( i \) will be put in a class by itself. For the remaining \( i \’s \), the distributions of \( x_i \’s \) are non-degenerate, and we use \( S_1, \ldots, S_g \) to denote the hidden classes formed by them. Numbers in the same class \( S_k \) are generated by linear functions of the same hidden random parameter \( z_k \). Different classes are governed by different random parameters. We know that the functions are linear, but no other information is given to us.

Let \( D_k \) denote the distribution of \( z_k \). There is a technical condition that is required of the \( D_k \’s \): there exists a constant \( \rho \in (0, 1) \) such that for every \( k \in [1, g] \) and every \( c \in \mathbb{R} \), \( \Pr[z_k = c] \leq 1 - \rho \). This condition says that \( D_k \) does not over-concentrate on any single value, which is quite a natural phenomenon. Our algorithm does not need to know \( \rho \).

\(^3\) There is a technical condition required of the input distribution to be explained in Section 2.

\(^4\) A less sophisticated method replaces \( n \log \log (mn) \) by \( mn \) which is beneficial for \( m = o(\log \log n) \).
2.1 Training phase

2.1.1 Learn the linear classes

We learn the classes and the linear functions using the first $3 \ln^2 n$ input instances. Denote these instances by $I_1, I_2, \cdots, I_{3 \ln^2 n}$. Let $x_i^{(a)}$ denote the $i$-th input number in $I_a$. We first recognize the degenerate distributions by checking which instances by $\epsilon$.

**Lemma 1.** Assume that $n \geq e^{2/(3 \rho)}$. It holds with probability at least $1 - 1/n$ that for all $i \in [1, n]$, if $x_i^{(a)}$ is the same for all $a \in [1, 3 \ln^2 n]$, the distribution of $x_i^{(a)}$ is degenerate.

**Proof.** Let $c_i$ be the observed value of $x_i^{(a)}$ for $a \in [1, 3 \ln^2 n]$. If the distribution of $x_i^{(a)}$ is not degenerate, the probability of $x_i^{(a)} = c_i$ for all $a \in [1, 3 \ln^2 n]$ is at most $(1 - \rho)^{3 \ln^2 n} \leq e^{-3 \rho \ln^2 n} \leq e^{-2 \ln n} = n^{-2}$. Applying the union bound establishes the lemma.

Assume that the degenerate distributions are taken out of the consideration. If $i$ and $j$ belong to the same class $S_k$, then $x_i^{(a)}$ and $x_j^{(a)}$ are linearly related as $a$ varies. Conversely, if $i$ and $j$ belong to different classes, it is highly unlikely that $x_i^{(a)}$ and $x_j^{(a)}$ remain linearly related as $a$ varies because they are governed by independent random parameters. We check if the triples of points $(x_i^{(a-2)}, x_j^{(a-2)})$, $(x_i^{(a-1)}, x_j^{(a-1)})$, and $(x_i^{(a)}, x_j^{(a)})$ are collinear for each $I_a, a \in [3, 3 \ln^2 n]$, and every distinct pair of $i$ and $j$ from $[1, n]$. We quantify this intuition in the following result.

**Lemma 2.** Let $i$ and $j$ be two distinct indices in $[1, n]$ that belong to different classes. For every $a \in [3, 3 \ln^2 n]$, let $E_{ij}^{(a)}$ denote the event that the points $(x_i^{(a-2)}, x_j^{(a-2)}), (x_i^{(a-1)}, x_j^{(a-1)})$, and $(x_i^{(a)}, x_j^{(a)})$ are not collinear. For any $n \geq e^{3/\rho^2}$, $\Pr \left[ \bigcup_{a=3}^{3 \ln^2 n} E_{ij}^{(a)} \right] \geq 1 - n^{-3}$.

**Proof.** We first prove a lower bound for $E_{ij}^{(3a)}$ for $a \in [1, \ln^2 n]$. It is well known [4] that the points $(x_i^{(3a-2)}, x_j^{(3a-2)}), (x_i^{(3a-1)}, x_j^{(3a-1)})$, and $(x_i^{(3a)}, x_j^{(3a)})$ are collinear if and only if

$$
\begin{vmatrix}
 x_i^{(3a-2)} & x_j^{(3a-2)} & 1 \\
 x_i^{(3a-1)} & x_j^{(3a-1)} & 1 \\
 x_i^{(3a)} & x_j^{(3a)} & 1 \\
\end{vmatrix} = 0.
$$

(1)

Assume that $x_i^{(3a-2)} = c_1$ and $x_i^{(3a-1)} = c_2$ for two fixed values $c_1$ and $c_2$. Since $i$ and $j$ are in different classes, $x_i^{(b)}$ and $x_j^{(b')}$ are independent for all $b$ and $b'$. Second, $x_j$ in one instance $I_b$ does not influence $x_j$ in a different instance $I_{b'}$. So there is no dependence among $x_i^{(3a)}$, $x_j^{(3a-2)}$, $x_j^{(3a-1)}$, and $x_j^{(3a)}$.

Suppose that $c_1 \neq c_2$. If $E_{ij}^{(3a)}$ does not occur, then by (1), we can express $x_j^{(3a)}$ as a function $f(c_1, c_2, x_i^{(3a)}, x_j^{(3a-2)}, x_j^{(3a-1)})$. Hence,

$$
\Pr \left[ E_{ij}^{(3a)} | x_i^{(3a-2)} = c_1 \land x_i^{(3a-1)} = c_2 \land c_1 \neq c_2 \right] = \sum_{c_3 \neq c_1, c_2} \Pr \left[ x_i^{(3a)} = c_3 \land x_j^{(3a-2)} = c'_1 \land x_j^{(3a-1)} = c'_2 \right] \cdot \Pr \left[ x_j^{(3a)} \neq f(c_1, c_2, c_3, c'_1, c'_2) \right] \geq \sum_{c_3 \neq c_1, c_2} \Pr \left[ x_i^{(3a)} = c_3 \land x_j^{(3a-2)} = c'_1 \land x_j^{(3a-1)} = c'_2 \right] \cdot \rho = \rho.
$$
If \(c_1 = c_2\), then (1) becomes \((x_i^{(3a)} - x_i^{(3a-1)})(x_i^{(3a-1)} - x_j^{(3a-2)}) = 0\). Thus,

\[
\Pr \left[ E_{ij}^{(3a)} \mid x_i^{(3a-2)} = c_1 \land x_j^{(3a-1)} = c_2 \land c_1 = c_2 \right] = \Pr \left[ x_j^{(3a-2)} \neq x_j^{(3a-1)} \right] \cdot \Pr \left[ x_i^{(3a)} = c_1 \right] \\
\geq \left( 1 - \Pr \left[ x_j^{(3a-2)} = x_j^{(3a-1)} \right] \right) \cdot \Pr \left[ x_i^{(3a)} = c_1 \right] \\
= \rho \cdot \left( 1 - \sum_c \Pr \left[ x_j^{(3a-2)} = c \right] \cdot \Pr \left[ x_j^{(3a-1)} = c \right] \right) \\
\geq \rho \cdot \left( 1 - (1 - \rho) \sum_c \Pr \left[ x_j^{(3a-2)} = c \right] \right) \\
= \rho^2.
\]

The above shows that the probability of \(E_{ij}^{(3a)}\) conditioned on some fixed values of \(x_i^{(3a-2)}\) and \(x_j^{(3a-1)}\) is at least \(\rho^2\). Hence, \(\Pr \left[ E_{ij}^{(3a)} \right] \geq \rho^2, \sum_{c_1,c_2} \Pr \left[ x_i^{(3a-2)} = c_1 \land x_i^{(3a-1)} = c_2 \right] = \rho^2\).

The events in \(\bigcup_{n=1}^{n} E_{ij}^{(3a)}\) are independent of each other. Therefore,

\[
\Pr \left[ \bigcup_{n=1}^{3 \ln^2 n} E_{ij}^{(a)} \right] \geq \Pr \left[ \bigcup_{n=1}^{ \ln^2 n} E_{ij}^{(3a)} \right] = 1 - \prod_{a=1}^{3 \ln^2 n} \Pr \left[ E_{ij}^{(a)} \right] \geq 1 - (1 - \rho^2)^{\ln^2 n}.
\]

Since \(n \geq e^{3/\rho^2}\), we get \((1 - \rho^2)^{\ln^2 n} \leq e^{-\rho^2} \leq n^{-3}\), establishing the lemma. \(\blacksquare\)

By Lemma 2, we keep a dictionary that stores \((i,j,b_{ij})\) for all \(i \neq j\) and \(i,j \in [1,n]\) such that the distributions of \(x_i\) and \(x_j\) are non-degenerate. Initially, \(b_{ij} = 1\) for all \((i,j)\). For each \(I_a\) where \(a \in [3,3 \ln^2 n]\), we perform the following. For every \((i,j)\), we check the event \(E_{ij}^{(a)}\) in \(O(1)\) time, set a bit variable \(\beta = 0\) if \(E_{ij}^{(a)}\) occurs and \(\beta = 1\) otherwise, and then update \(b_{ij} := b_{ij} \land \beta\). After going through all \(3 \ln^2 n\) input instances, we put \(x_i\) and \(x_j\) in the same class if and only if \(b_{ij} = 1\). By Lemmas 1 and 2 and the union bound, the classification is correct with probability at least \(1 - 1/n\). The processing time needed is \(O(n^2 \log^3 n)\).

### 2.1.2 Structures for the operation phase

After we obtain the classes, for each class \(S_k\), we fix an arbitrary index \(s_k \in S_k\). Then, we compute the equation of the line \(\ell_i\) that expresses \(x_i\) as a linear function in \(x_{s_k}\) for each \(i \in S_k \setminus \{s_k\}\). This can be done by picking any two input instances \(I_a\) and \(I_b\), and then computing the equation of the support line through \((x_{s_k}^{(a)}, x_i^{(a)})\) and \((x_{s_k}^{(b)}, x_i^{(b)})\) in \(O(1)\) time. The processing time needed over all classes is \(O(n)\).

Take another \(\ln n\) input instances. Sort all numbers in these instances into one sorted list \(L\). Form the \(V\)-list \((v_0, v_1, \cdots, v_{n+1})\), where \(v_0 = -\infty\), \(v_{n+1} = \infty\), and \(v_i\) has rank \(i\) in \(n\) in the list \(L\). The \(V\)-list requires \(O(n)\) space and can be computed in \(O(n \log^2 n)\) time. Note that if the distribution of \(x_i\) is degenerate, the same \(x_i\) appears \(n\) times in the sorted list \(L\), which implies that \(x_i\) must be selected to be an element of the \(V\)-list.

The \(V\)-list induces \(n\) horizontal lines at \(y\)-coordinates \(v_1, v_2, \cdots, v_n\). Let \(A_k\) denote the arrangement of the lines \(\ell_i\) computed for \(i \in S_k \setminus \{s_k\}\). We overlay these horizontal lines on top of \(A_k\). We draw vertical lines through the intersections between these horizontal lines and the lines in \(A_k\). We also draw vertical lines through the vertices of \(A_k\). The plane is divided into a set \(W\) of vertical slabs, where \(|W| = O(n|S_k|)\). Within each slab in \(W\), each line \(\ell_i\) in \(A_k\) lies strictly between two consecutive values \(v_r\) and \(v_{r+1}\), i.e., \(v_r\) is the predecessor of \(\ell_i\) in the \(V\)-list.
By a plane sweep over $A_k$ and the $n$ horizontal lines, we can figure out the predecessor of $\ell_i$ within each slab in $W$. For each slab in $W$, we store a list of the $\ell_i$'s in bottom-to-top order, and each line in the list stores its predecessor in the $V$-list. The lists for two consecutive slabs differ by either swapping two lines in $A_k$ or changing the predecessor of a line in $A_k$. Therefore, the $|W|$ lists can be stored in $O(|W| + |S_k|) = O(n|S_k|)$ space using a persistent lists data structure [5]. These persistent lists can be generated by a plane sweep over $A_k$ and the $n$ horizontal lines in $O(n|S_k| \log n)$ time.

We need to provide fast access to a particular slab in $W$ after specifying $x_{s_k}$. Take another $n^\varepsilon$ input instances for any choice of $\varepsilon \in (0, 1)$. Record the frequencies of $x_{s_k}$ falling into the slabs in $W$ among these $n^\varepsilon$ instances. We build a binary search tree on these slabs whose expected search time is asymptotically optimal with respect to the recorded frequencies. Let $T_k$ denote this asymptotically optimal binary search tree. There are $O(n|S_k|)$ nodes in $T_k$. At each node of $T_k$, we store the persistent list of lines in $A_k$ in bottom-to-top order within the slab corresponding to that node. The size of $T_k$ is $O(n|S_k|)$, and it can be constructed in $O(n|S_k|)$ time [6, 8]. Very low frequencies cannot give good estimate of the probability distribution of $x_{s_k}$, so navigating down $T_k$ to a node of very low frequency may be too time-consuming. Thus, if a search of $T_k$ reaches a node at depth below $\frac{1}{\varepsilon} \log_2 n$, we abort and perform a binary search among the slabs in $W$, which takes $O(\log |W|) = O(\log n)$ time.

The last ingredient is to allow the predecessor of $x_{s_k}$ in the $V$-list to be quickly located for all $k \in [1, g]$. We record the frequencies of $x_{s_k}$ falling to the intervals $[v_r, v_{r+1})$ among the $n^\varepsilon$ instances. Then, we build an asymptotically optimal binary search tree $\hat{T}_k$ with respect to these frequencies. The tree $\hat{T}_k$ uses $O(n)$ space, and it can be constructed in $O(n)$ time [6, 8]. As in the case of $T_k$, if a search of $\hat{T}_k$ reaches a node at depth below $\frac{1}{\varepsilon} \log_2 n$, we abort and perform a binary search in the $V$-list, which takes $O(\log n)$ time.

2.2 Operation phase

Given an input instance $I = (x_1, \ldots, x_n)$, for each class $S_k$, we query $T_k$ with $x_{s_k}$ to retrieve the sorted list $\sigma_k$ of numbers belonging to the class $S_k$. Precisely, $T_k$ gives fast access to the sorted sequence $\sigma_k \setminus \{x_{s_k}\}$, and then we spend $O(|\sigma_k|)$ time to insert $x_{s_k}$ into $\sigma_k \setminus \{x_{s_k}\}$. The numbers in $\sigma_k \setminus \{x_{s_k}\}$ are already stored with their predecessors in the $V$-list. We query $\hat{T}_k$ to obtain the predecessor of $x_{s_k}$ in the $V$-list.

Initialize an empty set $Z_r$ of lists for each interval $[v_r, v_{r+1})$. For each $x_i$ that is degenerately distributed, add $x_i$ to $Z_r$ where $v_r = x_i$. For each $k \in [1, g]$, if $\sigma_k \cap [v_r, v_{r+1})$ is non-empty, add $\sigma_k \cap [v_r, v_{r+1})$ to $Z_r$. Distributing $\sigma_k$ to the $Z_r$’s takes $O(|\sigma_k|) = O(|S_k|)$ time. Merge all lists in $Z_r$ into one sorted list. The merging is facilitated by a min-heap that stores the next element from each list in $Z_r$ to be considered for the next output number for the merged list. Thus, each step of the merging takes $O(\log |Z_r|)$ time. Finally, we concatenate in $O(n)$ time the merged lists for all $Z_r$’s to form the output sorted list.

Correctness is obvious. The limiting complexity has two main components. First, the sum of expected query times of $T_k$ and $\hat{T}_k$ for $k \in [1, g]$. Second, the total time spent on merging the lists in $Z_r$ for $r \in [0, n]$. The remaining processing time is $O(n + \sum_{k=1}^{g} |S_k|) = O(n)$. We give the analysis in the next section to show that the first two components sum to $O(n/\varepsilon + H_\pi/\varepsilon)$. Recall that $\pi(I)$ is the sequence of the ranks of numbers in $I$, which is a permutation of $[n]$, and $H_\pi$ is the entropy of the distribution of $\pi(I)$.
2.3 Analysis

Assign labels 0 to $n+1$ to $v_0, v_1, \ldots, v_n, v_{n+1}$ in this order. Similarly, assign labels $n+2$ to $2n+1$ to the input numbers $x_1, \ldots, x_n$ in this order.

Define the random variable $B^V$ to be the permutation of the labels that appear from left to right after sorting $\{v_0, \ldots, v_{n+1}\} \cup \{x_1, \ldots, x_n\}$ in increasing order.

For each $k \in [1, g]$, define a random variable $B^V_k$ to be the permutation of the labels that appear from left to right after performing the following operations: (1) sort $\{v_0, \ldots, v_{n+1}\} \cup \{x_i : i \in S_k \setminus \{s_k\}\}$ in increasing order, and (2) remove all $v_r$’s that do not immediately precede some $x_i$’s in the sorted list. Let $H^V_k$ denote the entropy of the distribution of $B^V_k$. Determining the sorted order $\sigma_k \setminus \{x_{s_k}\}$ and these numbers’ predecessors in the $V$-list takes at least $H^V_k$ expected time.

For each $k \in [1, g]$, define a random variable $\hat{B}^V_k$ to be the label of the predecessor of $x_{s_k}$ in the $V$-list. Let $\hat{H}^V_k$ denote the entropy of the distribution of $\hat{B}^V_k$. Determining the predecessor of $x_{s_k}$ in the $V$-list takes at least $\hat{H}^V_k$ expected time.

Our algorithm queries $T_k$ and $\hat{T}_k$ for $k \in [1, g]$, constructs $\sigma_k$ for $k \in [1, g]$ in $O(\sum_{k=1}^g |\sigma_k|)$ time, and then perform mergings in $O(n + \sum_{r=0}^n \sum_{k=1}^g |\sigma_k \cap [v_r, v_{r+1}]| \log |Z_r|)$ time. The additive $O(n)$ term takes care of every interval that contains only one $x_i$ that is degenerately distributed. Recall that $|Z_r|$ is the number of classes that have numbers falling into $[v_r, v_{r+1})$. If $T_k$ and $\hat{T}_k$ were the ideal binary search trees, querying them would take $H^V_k$ and $\hat{H}^V_k$ expected time, respectively. The total expected running time would then be

$$O\left(n + \sum_{k=1}^g H^V_k + \sum_{k=1}^g \hat{H}^V_k\right) + O\left(E \left[\sum_{r=0}^n \sum_{k=1}^g |\sigma_k \cap [v_r, v_{r+1}]| \log |Z_r|\right]\right).$$

(2)

We prove in the rest of this section that both $\sum_{k=1}^g H^V_k$ and $\sum_{k=1}^g \hat{H}^V_k$ are $O(n + H_x)$, and that $E[\sum_{r=0}^n \sum_{k=1}^g |\sigma_k \cap [v_r, v_{r+1}]| \log |Z_r|] = O(n)$. Moreover, although $T_k$ and $\hat{T}_k$ are not ideal binary search trees, their expected query complexities are $O(H^V_k/\varepsilon)$ and $O(\hat{H}^V_k/\varepsilon)$, respectively, as shown in [1, Lemma 3.4]. Therefore, the total expected running time is $O(n/\varepsilon + H_x/\varepsilon)$.

We need two technical results.

► Lemma 3 ([3, Theorem 2.5.1]). Let $H(X_1, \ldots, X_n)$ be the joint entropy of independent random variables $X_1, \ldots, X_n$. Then $H(X_1, \ldots, X_n) = \sum_{i=1}^n H(X_i)$.

► Lemma 4 ([1, Lemma 2.3]). Let $X : \mathcal{U} \to \mathcal{X}$ and $Y : \mathcal{U} \to \mathcal{Y}$ be two random variables obtained with respect to the same arbitrary distribution over the universe $\mathcal{U}$. Suppose that the function $f : (I, X(I)) \mapsto Y(I)$, $I \in \mathcal{U}$, can be computed by a comparison-based algorithm with $C$ expected comparisons, where the expectation is over the distribution on $\mathcal{U}$. Then, $H(Y) \leq C + O(H(X))$.

We show that both $\sum_{k=1}^g H^V_k$ and $\sum_{k=1}^g \hat{H}^V_k$ are $O(n + H_x)$.

► Lemma 5.

(a) $\sum_{k=1}^g H^V_k = O\left(n + H(B^V)\right) = O\left(n + H_x\right)$,

(b) $\sum_{k=1}^g \hat{H}^V_k = O\left(n + H_x\right)$.

Proof. Consider (a). Suppose that we are given a setting of $B^V$, i.e., the permutation of labels from left to right in the sorted order of $\{v_0, \ldots, v_{n+1}\} \cup \{x_1, \ldots, x_n\}$. We scan the sorted list from left to right. We maintain the most recently scanned $v_r$. Suppose that we see a number $x_i$. Let $S_k$ be the class to which $x_i$ belongs. If this is the first time that we
encounter an index in $S_k$, we initialize an output list for $B_k^V$ that contains the label of $v_r$ followed by the label of $x_i$. If this is not the first time that we encounter an index in $S_k$, we append the label of $x_i$ to the output list for $B_k^V$. There is an exception that when $i = s_k$; we do not output the label of $x_i$. Clearly, we obtain the settings of all $B_k^V$’s correctly from $B^V$. The number of comparisons needed is $O(n)$. Therefore, Lemmas 3 and 4 imply that

$$\sum_{k=1}^{g} H_k^V = H(B_1^V, \cdots, B_g^V) = O(n + H(B^V)).$$

Given $(I, \pi(I))$, we use $\pi(I)$ to sort $I$ and then merge the sorted order with $(v_0, \cdots, v_{n+1})$. Afterwards, we scan the sorted list to output the labels of the numbers. This gives the setting of $B^V$. Clearly, $O(n)$ comparisons suffice, and so Lemma 4 implies that $H(B^V) = O(n + H_n)$. This completes the proof of (a).

The settings of $\hat{B}_1^V, \cdots, \hat{B}_g^V$ can be derived similarly by using $\pi(I)$ to sort $I$, merging the sorted sequence with $(v_0, \cdots, v_{n+1})$, and then scanning the merged sequence. Then, Lemmas 3 and 4 imply that $\sum_{k=1}^{g} \hat{H}_k^V = H(\hat{B}_1^V, \cdots, \hat{B}_g^V) = O(n + H_n)$, establishing (b).

We will show that it holds with high probability that $E[|Z_r|] = O(1)$ for all $r \in [0, n]$ simultaneously. It implies that $E[\max_{r \in [0, n]} |Z_r|] = O(1)$ with high probability. Then,

$$E \left[ \sum_{r=0}^{n} |\sigma_k \cap [v_r, v_{r+1})| \log |Z_r| \right] \leq E \left[ \max_{r \in [0, n]} |Z_r| \cdot \sum_{r=0}^{n} |\sigma_k \cap [v_r, v_{r+1})| \right]$$

$$= |\sigma_k| \cdot E \left[ \max_{r \in [0, n]} |Z_r| \right]$$

$$= O(|\sigma_k|).$$

Hence,

$$E \left[ \sum_{r=0}^{n} \sum_{k=1}^{g} |\sigma_k \cap [v_r, v_{r+1})| \log |Z_r| \right] = \sum_{k=1}^{g} E \left[ \sum_{r=0}^{n} |\sigma_k \cap [v_r, v_{r+1})| \log |Z_r| \right]$$

$$\leq O \left( \sum_{k=1}^{g} |\sigma_k| \right)$$

$$= O(n).$$

The second term in (2) can thus be replaced by $O(n)$.

Our proof of $E[|Z_r|] = O(1)$ for all $r \in [0, n]$ with high probability is modeled after the proof of a similar result in [1]. There is a small twist due to the handling of the classification.

Lemma 6. It holds with probability at least $1 - O(1/n)$ that for all $r \in [0, n]$, $E[|Z_r|] = O(1)$.

Proof. Let $I_1, \cdots, I_{ln} n$ denote the input instances used in the training phase for building the $V$-list. Let $y_1, y_2, \cdots, y_{ln} n$ denote the sequence formed by concatenating $I_1, \cdots, I_{ln} n$ in this order. We adopt the notation that for each $\alpha \in [1, ln n]$, $y_\alpha$ belongs to the class $S_{k_\alpha}$ and the input instance $I_{k_\alpha}$.

Fix any distinct index pairs $\alpha, \beta \in [1, ln n]$ such that $y_\alpha \leq y_\beta$. Let $J^\beta_\alpha$ be the set of index pairs $\{(a, k) : a \in [1, ln n], k \in [1, g]\} \setminus \{(a, k_\alpha), (a, k_\beta)\}$. For any $(a, k) \in J^\beta_\alpha$, let $Y^{\beta_\alpha}(a, k)$ be an indicator random variable such that if some element of the input instance $I_a$, $a \in [1, ln n]$, belongs to $S_k$ and falls into $(y_\alpha, y_\beta)$, then $Y^{\beta_\alpha}(a, k) = 1$; otherwise, $Y^{\beta_\alpha}(a, k) = 0$. Define $Y^{\beta_\alpha} = \sum_{(a, k) \in J^\beta_\alpha} Y^{\beta_\alpha}(a, k)$.

Among the $(a, k)$’s in $J^\beta_\alpha$, the random variables $Y^{\beta_\alpha}(a, k)$ are independent from each other. By Chernoff’s bound, for any $\mu \in [0, 1]$,

$$\Pr \left[ Y^{\beta_\alpha} \leq (1 - \mu)E[Y^{\beta_\alpha}] \right] \leq e^{-\mu^2E[Y^{\beta_\alpha}]}/2.$$
Setting \( \mu = \sqrt{35} - 5 \approx 0.9161 \) shows that if \( \mathbb{E}[Y^\beta_\alpha] > \frac{1}{6 - \sqrt{35}} \ln n \), then \( Y^\beta_\alpha > \ln n \) with probability at least \( 1 - n^{-5} \). Since the above statement holds for any fixed choices of \( \alpha \) and \( \beta \) such that \( y_\alpha \leq y_\beta \), we can apply the union bound to the \( O(n^2 \log^2 n) \) possible choices of \( \alpha \) and \( \beta \) and conclude that:

It holds with probability at least \( 1 - O(n^{-2}) \) that for any distinct index pairs \( \alpha, \beta \in \{1, n \ln n\} \) such that \( y_\alpha \leq y_\beta \), if \( \mathbb{E}[Y^\beta_\alpha] > \frac{1}{6 - \sqrt{35}} \ln n \), then \( Y^\beta_\alpha > \ln n \).

For every \( r \in [0, n + 1] \), let \( y_{\alpha_r} \) denote \( v_r \), where \( y_{\alpha_0} = -\infty \) and \( y_{\alpha_{n+1}} = \infty \). Fix a particular \( r \in [0, n + 1] \). By construction, there are \( \ln n \) numbers among \( I_1, \ldots, I_{\ln n} \) that fall in \( [v_r, v_{r+1}] \), which guarantees the event of \( Y^\alpha_{\alpha_r+1} \leq \ln n \). Our previous conclusion implies that \( \mathbb{E}[Y^\alpha_{\alpha_r+1}] \leq \frac{1}{6 - \sqrt{35}} \ln n \) with probability at least \( 1 - O(n^{-2}) \).

We relate \( \mathbb{E}[Y^\alpha_{\alpha_r+1}] \) to \( \mathbb{E}[|Z_r|] \) as follows. Let \( X_{kr} \) be the indicator random variable such that if some element of the input instance belongs to \( S_k \) and falls into \( [v_r, v_{r+1}] \), then \( X_{kr} = 1 \); otherwise, \( X_{kr} = 0 \). Then \( \sum_{k=1}^{\ln n} X_{kr} = |Z_r| \), implying that \( \sum_{k=1}^{\ln n} \mathbb{E}[X_{kr}] = \mathbb{E}[|Z_r|] \). The random process that generates the input instances is oblivious of the training phase. It follows that \( \mathbb{E}[Y^\alpha_{\alpha_r+1}] \) should be the same as \( \sum_{k=1}^{\ln n} \sum_{a=1}^{g} \mathbb{E}[X_{kr}] \), except that the index pairs \( (\alpha_r, k_a) \) and \( (\alpha_{r+1}, k_{a+1}) \) are excluded from \( J^{\alpha_r+1} \), but these two cases are considered in \( \sum_{a=1}^{\ln n} \mathbb{E}[X_{kr}] \). Therefore,

\[
\mathbb{E}[Y^\alpha_{\alpha_r+1}] \geq \left( \sum_{a=1}^{\ln n} \sum_{k=1}^{g} \mathbb{E}[X_{kr}] \right) - 2 = \ln n \cdot \mathbb{E}[|Z_r|] - 2. \tag{3}
\]

We have shown previously that \( \mathbb{E}[Y^\alpha_{\alpha_r+1}] \leq \frac{1}{6 - \sqrt{35}} \ln n \) with probability at least \( 1 - O(n^{-2}) \). It follows that \( \mathbb{E}[|Z_r|] = O(1) \) with probability at least \( 1 - O(n^{-2}) \). Since the above statement holds for every fixed \( r \in [0, n] \), by the union bound, it holds with probability at least \( 1 - O(1/n) \) that \( \mathbb{E}[|Z_r|] = O(1) \) for all \( r \in [0, n] \).

It remains to show that the expected query complexities of \( T_k \) and \( \hat{T}_k \) are \( O(H_k^V/\varepsilon) \) and \( O(H_k^V/\varepsilon) \), respectively. The argument is based on Chernoff’s bound and the fact that if in a search in \( T_k \) or \( \hat{T}_k \) reaches a pruned node, it means that the search requires \( \Omega(\varepsilon \log n) \) time. The exact same arguments have been made by Ailon et al. [1, Lemma 3.4].

\textbf{Theorem 7.} For any \( \varepsilon \in (0, 1) \), there exists a self-improving sorter of \( O(n \varepsilon + H_k^V) \) limiting complexity for any product distribution with hidden linear classes. The storage needed is \( O(n^3) \). The training phase processes \( O(n^4) \) input instances in \( O(n^4 \log^5 n) \) time, and it succeeds with probability at least \( 1 - 1/n \).

\section{Mixture of product distributions}

Let \( \kappa \) be the number of product distributions in the mixture. Although \( \kappa \) is hidden, we are given an upper bound \( m \geq \kappa \). Let \( D_q, q \in [1, \kappa] \), be the hidden product distributions in the mixture. In each \( D_q \), the \( i \)-th input number is drawn from \( D_{q,i} \), i.e., \( D_q = \prod_{i=1}^{n} D_{q,i} \). The input distribution is \( \sum_{q=1}^{\kappa} \lambda_q D_q \) for some hidden positive \( \lambda_q \)'s such that \( \sum_{q=1}^{\infty} \lambda_q = 1 \).

\subsection{Training phase}

Take \( m (\ln m + \ln n) \) input instances and sort all of these numbers in increasing order. Select the numbers in the sorted list of ranks \( \ln m + \ln n, 2(\ln m + \ln n), \ldots, mn (\ln m + \ln n) \). The selected numbers induce a doubly linked list \( V \) of intervals: \( (-\infty, v_1), [v_1, v_2), \ldots, [v_mn, \infty) \).
We first show that sorting all \( [v_r, v_{r+1}) \) for \( r \in [0, mn] \), where \( v_0 = -\infty, v_{mn+1} = \infty \), and we abuse the notation slightly to take \( [v_0, v_1) \) to mean \(( -\infty, v_1)\).

We organize a balanced binary search tree \( T^V \) whose nodes correspond to intervals in \( V \).

Use another \( O(m^2 n^2) \) input instances to record the frequency \( f_{v_r} \) that \( x_i \) falls into \([v_r, v_{r+1})\). Then, for every \( i \in [1, n] \), build an asymptotically optimal binary search tree \( T_i \) with respect to the \( f_{v_r} \)'s on the intervals with positive frequencies. This can be done in \( O(m^2 n^2) \) time \([6, 8]\). The size of \( T_i \) is \( O(m^2 n^2) \). If a search of \( T_i \) reaches a node at depth below \( \frac{1}{2} \log_2(mn) \) or is unsuccessful, we answer the query by searching \( T^V \) which takes \( O(\log(mn)) \) time.

We also need a fast dictionary data structure that can be built in \( O(mn) \) time and space. But we defer its description until we explain the need for it in the operation phase.

The total space required is \( O(mn + m^2 n^{1+\varepsilon}) \). The total time spent in the training phase is \( O(mn(\log m + \log n)^2 + m^2 n^{1+\varepsilon}) \).

### 3.2 Operation phase

We first give a naive method that is slow to illustrate the overall strategy. Given an input instance \( I = (x_1, \cdots, x_n) \), for each \( i \in [1, n] \), we search \( T_i \) to place \( x_i \) in the interval \([v_r, v_{r+1})\) that contains it. For each \( r \in [0, mn] \), the entry \([v_r, v_{r+1}) \) in \( V \) keeps a list \( N_r \) of \( x_i \)'s that fall into it. We sort each \( N_r \) in \( O(|N_r| \log |N_r|) \) time. Then, we concatenate the sorted \( N_r \)'s in increasing order of \( r \) to form the output sorted list.

Let \( t_i \) denote the expected time to query \( T_i \) with \( x_i \). Assume that we can prove as in \([1]\) that \( \text{E}[|N_r|^2] = O(1) \). Then, sorting each \( N_r \) takes only \( O(1) \) expected time. Therefore, the total time for processing \( I \) is \( O(mn + \sum_{i=1}^n t_i) \). This is too slow unless \( m = o(\log n) \). The \( O(mn) \) term arises from scanning the list \( V \) in order to concatenate the sorted \( N_r \)'s in the right order. However, at most \( n \) of these \( mn + 1 \) intervals are “useful” because there are only \( n \) input numbers. We describe an improvement below.

We maintain a dictionary \( U \) that is initially empty. For each \( i \in [1, n] \), \( T_i \) leads us to the interval \([v_r, v_{r+1})\) that contains \( x_i \). We find \( v_r \) in \( U \). If \( v_r \) is present in \( U \), we simply add \( x_i \) to \( N_r \). Otherwise, we insert \( v_r \) to \( U \) and initialize \( N_r \) to contain \( x_i \) alone. After seeing all \( n \) input numbers, we find the minimum element in \( U \) and then find successors iteratively. This allows us to visit the non-empty \( N_r \)'s in increasing order of \( r \). So we can concatenate the sorted \( N_r \)'s in \( O(n) \) time. At the end, we delete all elements from \( U \) in preparation for sorting the next input instance.

The van Emde Boas tree \([9]\) supports dictionary operations in \( O(\log \log N) \) worst-case time each, where \( N \) is the size of the universe. It means \( O(\log \log(mn)) \) time in our case. In the training phase, we construct a van Emde Boas tree with universe \( V \). It uses \( O(mn) \) space and can be built in \( O(mn) \) time.\(^5\) The asymptotical storage and processing time required by the training phase is unaffected.

In all, the running time is reduced to \( O(n \log \log(mn) + \sum_{i=1}^n t_i) \).

### 3.3 Analysis

We first show that sorting all \( N_r \)'s takes \( O(n) \) expected time.

\[ \textbf{Lemma 8.} \text{ It holds with probability at least } 1 - 1/n \text{ that } \text{E}[\sum_{r=0}^{mn} |N_r| \log |N_r|] = O(n). \]

\(^5\) The space usage according to the description in \([9]\) is \( O(mn \log \log(mn)) \), but it can be improved to \( O(mn) \) as mentioned in \([7]\).
Proof. We first prove that \( E[|N_r|] = O(1) \) for all \( r \in [0, mn] \) are satisfied simultaneously with probability at least \( 1 - 1/n \).

As a shorthand, let \( \gamma = \ln m + \ln n \). Let \( I_1, \ldots, I_m \) denote the input instances used in the training phase for building the list \( V \). Let \( y_1, y_2, \ldots, y_{mn} \) denote the sequence formed by concatenating \( I_1, \ldots, I_m \), in this order. We adopt the notation that for each \( \alpha \in [1, mn] \), \( y_\alpha \) belongs to \( I_\alpha \), and \( y_\alpha \) is drawn from \( D_{y_\alpha, I_\alpha} \).

Fix any distinct index pairs \( \alpha, \beta \in [1, mn] \) such that \( y_\alpha \leq y_\beta \). For every \( q \in [1, \kappa] \), let \( J_\alpha^q(q) \) be the set of index triplets \( \{(a, q, i) : a \in [1, m|\gamma|], i \in [1, n]\} \setminus \{(a_\alpha, q_\alpha, i_\alpha), (a_\beta, q_\beta, i_\beta)\} \). For any \( (a, q, i) \in J_\alpha^q(q) \), let \( Y_\alpha^q(a, q, i) \) be the indicator random variable such that if \( I_a \sim D_q \) and \( x_i \) in \( I_a \) falls into \( [y_\alpha, y_\beta] \), then \( Y_\alpha^q(a, q, i) = 1 \); otherwise, \( Y_\alpha^q(a, q, i) = 0 \). Define \( Y_\alpha^q(q) = \sum_{(a, q, i) \in J_\alpha^q(q)} Y_\alpha^q(a, q, i) \).

Among the \( (a, q, i) \)'s in \( J_\alpha^q(q) \), the random variables \( Y_\alpha^q(a, q, i) \)'s are independent from each other. By Chernoff's bound, for any \( \mu \in [0, 1] \), \( \Pr \left[ Y_\alpha^q(q) \leq (1 - \mu)E[Y_\alpha^q(q)] \right] \leq e^{-\mu^2E[Y_\alpha^q(q)]/2} \). Setting \( \mu = \sqrt{35} - 5 \approx 0.9161 \) shows that if \( E[Y_\alpha^q(q)] > \frac{1}{6-\sqrt{35}} \gamma \), then \( Y_\alpha^q(q) \geq \gamma \) with probability at least \( 1 - m^{-\kappa}n^{-5} \). Since the above statement holds for any fixed choices of \( q, \alpha \) and \( \beta \) such that \( y_\alpha \leq y_\beta \), we can apply the union bound to the \( O(kmn^2\log m + \log n)^2 \) possible choices of \( q, \alpha \) and \( \beta \) and conclude that:

It holds with probability at least \( 1 - O(m^{-1}n^{-2}) \) that for any \( q \in [1, \kappa] \) and any \( \alpha, \beta \in [1, mn] \) such that \( y_\alpha \leq y_\beta \), if \( E[Y_\alpha^q(q)] > \frac{1}{6-\sqrt{35}} \gamma \), then \( Y_\alpha^q(q) > \gamma \).

For every \( r \in [0, mn] \), let \( y_{\alpha r} \) denote \( y_{r+1} \), where \( y_{oo} = -\infty \) and \( y_{o\alpha o+1} = \infty \). Fix a particular \( r \in [0, mn] \). By construction, there are \( \gamma \) numbers among \( I_1, \ldots, I_m \) that fall in \( [v_r, v_{r+1}] \), which guarantees the event of \( Y_{\alpha o}^{\gamma+1}(q) \leq \gamma \) for all \( q \in [1, \kappa] \). By our previous conclusion, it holds with probability at least \( 1 - O(m^{-1}n^{-2}) \) that \( E[Y_{\alpha o}^{\gamma+1}(q)] \leq \frac{1}{6-\sqrt{35}} \gamma \) for all \( q \in [1, \kappa] \). Let \( Y_{\alpha o}^{\gamma+1} = \sum_{q=0}^{\kappa} Y_{\alpha o}^{\gamma+1}(q) \). It follows that:

\[
E[Y_{\alpha o}^{\gamma+1}] = O(\kappa \gamma) \quad \text{holds with probability at least} \quad 1 - O(m^{-1}n^{-2}).
\] (4)

Let \( X_{ir} \) be the indicator random variable such that if \( x_i \) falls into the interval \( [v_r, v_{r+1}] \), then \( X_{ir} = 1 \); otherwise, \( X_{ir} = 0 \). Then \( \sum_{i=1}^{n} X_{ir} = |N_r| \). Note that \( Y_{\alpha o}^{\gamma+1} \) counts every \( x_i \)'s in \( I_a \) that falls into \( [v_r, v_{r+1}] \), except for the two cases of \( (a, i) = (a_\alpha, i_\alpha) \land I_a \sim D_{y_\alpha} \), and \( (a, i) = (a_{\alpha+1}, i_{\alpha+1}) \land I_a \sim D_{y_\beta} \). The random process that generates the input is oblivious of the training phase. Therefore, \( E[Y_{\alpha o}^{\gamma+1}] \) is expected to be the same as \( \sum_{q=1}^{\kappa} \sum_{i=1}^{n} E[X_{ir}] \), except that the cases of \( (a, i) = (a_\alpha, i_\alpha) \land I_a \sim D_{y_\alpha} \), and \( (a, i) = (a_{\alpha+1}, i_{\alpha+1}) \land I_a \sim D_{y_\beta} \), are excluded from \( J_{\alpha o}^{\gamma+1} \), but these two cases are considered in \( \sum_{q=1}^{\kappa} \sum_{i=1}^{n} E[X_{ir}] \). Hence,

\[
E[Y_{\alpha o}^{\gamma+1}] \geq \left( m_{\gamma} \cdot \sum_{i=1}^{n} E[X_{ir}] \right) - 2 = (m_{\gamma} \cdot E[|N_r|]) - 2.
\]

Substituting (4) into the above inequality shows that \( E[|N_r|] = O(1) \). The \( O(1) \) bounds on \( E[|N_r|] \) hold for a fixed \( r \) with probability at least \( 1 - O(m^{-1}n^{-2}) \). Applying the union bound over \( r \in [0, mn] \) establishes the claim that \( E[|N_r|] = O(1) \) for all \( r \in [0, mn] \) are satisfied simultaneously with probability at least \( 1 - 1/n \). It follows that \( E[\max_{r \in [0, mn]} |N_r|] = O(1) \) with probability at least \( 1 - 1/n \).

The expected total time to sort the \( N_r \)'s is

\[
E \left[ \sum_{r=0}^{mn} |N_r| \log |N_r| \right] = E \left[ \sum_{r=0}^{mn} \sum_{i=1}^{n} X_{ir} \log |N_r| \right] \leq \sum_{i=1}^{n} E \left[ \max_{r \in [0, mn]} |N_r| \cdot \sum_{r=0}^{mn} X_{ir} \right].
\]

Observe that \( \sum_{r=0}^{mn} X_{ir} = 1 \) because \( x_i \) falls into exactly one of the \( mn + 1 \) intervals. As a result, it holds with probability at least \( 1 - 1/n \) that \( E[\sum_{r=0}^{mn} |N_r| \log |N_r|] = O(n) \).
Next, we bound $\sum_{i=1}^n t_i$. Let $\mu_{iqr}$ denote the probability of $x_i \in [v_r,v_{r+1})$ conditioned on $x_i \sim D_{q,i}$. Define $\mu_{i}$ to be the the probability of $x_i \in [v_r,v_{r+1})$, and therefore, $\mu_{i} = \sum_{q=1}^\kappa \lambda_q \mu_{iqr}$.

Let $H^V_i$ be the entropy of the distribution of the predecessor of $x_i$ in $V$. So $H^V_i = \sum_{r=0}^{m} \mu_{i} \log (1/\mu_{i})$. As shown in [1, Lemma 3.4], $T_i$ has an expected search time of

$$O\left(\frac{H^V_i}{\varepsilon}\right) = O\left(\frac{1}{\varepsilon} \sum_{r=0}^{m} \mu_{i} \log (1/\mu_{i})\right) = O\left(\frac{1}{\varepsilon} \sum_{r=0}^{m} \left(\sum_{q=1}^\kappa \lambda_q \mu_{iqr}\right) \log \left(\frac{1}{\sum_{q=1}^\kappa \lambda_q \mu_{iqr}}\right)\right).$$

Observe that $\log \left(\frac{1}{\sum_{q=1}^\kappa \lambda_q \mu_{iqr}}\right) \leq \log \frac{1}{\lambda_q \mu_{iqr}}$ for any $q$. Therefore,

$$H^V_i \leq \sum_{q=1}^\kappa \sum_{r=0}^{m} \lambda_q \mu_{iqr} \log \frac{1}{\lambda_q \mu_{iqr}} = \sum_{q=1}^\kappa \sum_{r=0}^{m} \lambda_q \mu_{iqr} \log \frac{1}{\lambda_q} + \sum_{q=1}^\kappa \sum_{r=0}^{m} \lambda_q \mu_{iqr} \log \frac{1}{\mu_{iqr}}.$$

Note that $\sum_{r=0}^{m} \lambda_q \mu_{iqr} = \lambda_q$ as $\sum_{r=0}^{m} \mu_{iqr} = 1$. Then, $\sum_{q=1}^\kappa \sum_{r=0}^{m} \lambda_q \mu_{iqr} \log (1/\lambda_q) = \sum_{q=1}^\kappa \lambda_q \log (1/\lambda_q)$, which is at most $\log \kappa$. Then,

$$\sum_{i=1}^n t_i = O\left(\frac{n}{\varepsilon} \log \kappa\right) + O\left(\frac{1}{\varepsilon} \sum_{i=1}^n \sum_{q=1}^\kappa \sum_{r=0}^{m} \lambda_q \mu_{iqr} \log \frac{1}{\mu_{iqr}}\right) = O\left(\frac{n}{\varepsilon} \log \kappa\right) + O\left(\frac{1}{\varepsilon} \sum_{q=1}^\kappa \lambda_q \left(\sum_{i=1}^n \sum_{r=0}^{m} \mu_{iqr} \log \frac{1}{\mu_{iqr}}\right)\right).$$

Let $H^V_{q,i} = \sum_{r=0}^{m} \mu_{iqr} \log (1/\mu_{iqr})$, the entropy of the distribution of the predecessor of $x_i$ in $V$ conditioned on $x_i \sim D_{q,i}$. Then, $\sum_{i=1}^n \sum_{r=0}^{m} \mu_{iqr} \log (1/\mu_{iqr}) = \sum_{i=1}^n H^V_{q,i}$. By Lemma 5(b) and setting $g = n$, we obtain $\sum_{i=1}^n H^V_{q,i} = O(n + H_{\pi,q})$, where $H_{\pi,q}$ is the entropy of $\pi(I)$ conditioned on $I \sim D_q$. It is well-known that an unconditional entropy is greater than or equal to its conditioned counterpart, so $H_{\pi} \geq H_{\pi,q}$. Therefore, $\sum_{i=1}^n H^V_{q,i} = O(n + H_{\pi})$.

Thus, $\sum_{i=1}^n t_i = O\left(\frac{n}{\varepsilon} \log \kappa + \frac{1}{\varepsilon} \sum_{q=1}^\kappa \lambda_q (n + H_{\pi})\right) = O\left(\frac{n}{\varepsilon} \log \kappa + H_{\pi}/\varepsilon\right)$.

**Theorem 9.** For any constant $\varepsilon > 0$, there exists a self-improving sorter of limiting complexity $O(n \log \log (mn) + (n/\varepsilon) \log \kappa + H_{\pi}/\varepsilon)$ for any hidden mixture of $\kappa$ product distribution. The parameter $\kappa$ is hidden, but an upper bound $m \geq \kappa$ is given. The storage needed is $O(mn + m^2 n^{1+\varepsilon})$. The training phase processes $O(m \log m + \log n) + m\varepsilon n^2$ input instances in $O(mn \log (m + \log n)^2 + m^2 n^{1+\varepsilon})$ time, and it succeeds with probability at least $1 - 1/n$.

4 Conclusion

There are several possible directions for future research. One is to extend the hidden classification to allow the $x_i$’s in the same class $S_k$ to be some fixed-degree polynomial in the random parameter $z_k$. Linear functions in $z_k$ have the nice property that any $x_i$ and
$x_j$ in the same class are linearly related. This helps us to learn the hidden classes. We lose this property in the case of fixed-degree polynomials. Another direction is to improve the limiting complexity in the case of a hidden mixture of product distributions. Can the term $O(n \log \log (mn) + (n/\varepsilon) \log \kappa)$ be reduced? If the upper bound $m$ of $\kappa$ is not too far off, then $n \log \log (mn) \approx n \log \log \kappa + n \log \log n$, which means that our limiting complexity becomes $O(n \log \log n + (n/\varepsilon) \log \kappa + H(\pi/\varepsilon))$. Although $n \log \log n$ is $o(n \log n)$, it would be nice to eliminate it or reduce it further. It is also unclear whether the factor $\log \kappa$ is necessary.

It would also be interesting to design self-improving algorithms for other problems and possibly other input settings as well.

References

Online Scheduling of Car-Sharing Requests Between Two Locations with Many Cars and Flexible Advance Bookings

Kelin Luo
School of Management, Xi’an Jiaotong University, Xi’an, China
luokelin@xjtu.edu.cn

Thomas Erlebach
Department of Informatics, University of Leicester, Leicester, United Kingdom
t.erlebach@leicester.ac.uk

Yinfeng Xu
School of Management, Xi’an Jiaotong University, Xi’an, China
yfxu@xjtu.edu.cn

Abstract
We study an on-line scheduling problem that is motivated by applications such as car-sharing, in which users submit ride requests, and the scheduler aims to accept requests of maximum total profit using $k$ servers (cars). Each ride request specifies the pick-up time and the pick-up location (among two locations, with the other location being the destination). The scheduler has to decide whether or not to accept a request immediately at the time when the request is submitted (booking time). We consider two variants of the problem with respect to constraints on the booking time: In the fixed booking time variant, a request must be submitted a fixed amount of time before the pick-up time. In the variable booking time variant, a request can be submitted at any time during a certain time interval (called the booking horizon) that precedes the pick-up time. We present lower bounds on the competitive ratio for both variants and propose a balanced greedy algorithm (BGA) that achieves the best possible competitive ratio. We prove that, for the fixed booking time variant, BGA is 1.5-competitive if $k = 3i$ ($i \in \mathbb{N}$) and the fixed booking length is not less than the travel time between the two locations; for the variable booking time variant, BGA is 1.5-competitive if $k = 3i$ ($i \in \mathbb{N}$) and the length of the booking horizon is less than the travel time between the two locations; and BGA is 5/3-competitive if $k = 5i$ ($i \in \mathbb{N}$) and the length of the booking horizon is not less than the travel time between the two locations.

2012 ACM Subject Classification Theory of computation → Online algorithms

Keywords and phrases Car-sharing system, Competitive analysis, On-line scheduling

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.64
1 Introduction

In a car-sharing system, a company offers cars to customers for a period of time. Customers can pick up a car in one location, drive it to another location, and return it there. Car booking requests arrive on-line, and the goal is to maximize the profit obtained from satisfied requests. We refer to this problem as the car-sharing problem.

In a real setting, customer requests for car bookings arrive over time, and the decision about each request must be made immediately, without knowledge of future requests. This gives rise to an on-line problem that bears some resemblance to interval scheduling, but in which additionally the pick-up and drop-off locations play an important role: The server that serves a request must be at the pick-up location at the start time of the request and will be located at the drop-off location at the end time of the request. We consider a setting where all driving routes go between two fixed locations, but can be in either direction. For example, the two locations could be a residential area and a nearby shopping mall or central business district. Other applications that provide motivation for the problems we study include car rental, taxi dispatching and boat rental for river crossings. A server can serve two consecutive requests only if the drop-off location of the first request is the same as the pick-up location of the second request, or if there is enough time to travel between the two locations otherwise. We allow empty movements, i.e., a server can be moved from one location to another while not serving a request. Such empty movements could be implemented by having company staff drive a car from one location to another, or in the future by self-driving cars.

With respect to constraints on the booking time, one can consider the fixed booking time variant and the variable booking time variant of the car-sharing problem [7]. The fixed booking time variant requires users to submit requests in such a way that the amount of time between the booking time of a request and its start time is a fixed value, independent of the request. This simplifies the scheduling task because the order of the start times of the requests is the same as the order of their release times (booking times). It is, however, less convenient for users because they have to book a request at a specific time. In the variable booking time variant, the booking time of a request must lie in a certain time interval (called the booking horizon) before the start time of the request. Users can book a request at any time in this interval.

1.1 Related Work

In [7], the authors studied the car-sharing problem for the special case of two locations and a single server, considering both fixed booking times and variable booking times, and presented tight results for the competitive ratio. The optimal competitive ratio was shown to be 2 for fixed booking times and 3 for variable booking times. In [8], the authors dealt with the car-sharing problem with two locations and two servers, considering only the case of fixed booking times, and presented tight results for the competitive ratio. The optimal competitive ratio was shown to be 2. In contrast to the previous work on car-sharing between two locations, in this paper we consider the car-sharing problem for both fixed booking times and variable booking times in the setting with $k$ servers where $k$ can be arbitrarily large. As a larger number of servers provides more flexibility to the algorithm, different lower bound constructions and different techniques for analyzing the competitive ratio of an algorithm are required. It seems natural to expect that a large number of servers can help an algorithm to achieve better competitive ratio, but our results show that, surprisingly, 3 servers (in one case) and 5 servers (in another case) already allow us to get the best competitive ratio, and no improvement is possible with more servers.
Böhmlová et al. [3] showed that if all customer requests for car bookings are known in advance, the problem of maximizing the number of accepted requests is solvable in polynomial time. Furthermore, they considered the problem variant with two locations where each customer requests two rides (in opposite directions) and the scheduler must accept either both or neither of the two. They proved that this variant is NP-hard and APX-hard. In contrast to their work, we consider the on-line version of the problem with \( k \) servers.

Amongst other related work, the problem that is closest to our setting is the on-line dial-a-ride problem (OLDARP). In OLDARP, transportation requests between locations in a metric space arrive over time, but typically it is assumed that requests want to be served “as soon as possible” rather than at a specific time as in our problem. Versions of OLDARP with the objective of serving all requests while minimizing the makespan [1, 2] or the maximum flow time [6] have been widely studied in the literature. The versions of OLDARP where not all requests need to be served includes competitive algorithms for requests with deadlines where each request must be served before its deadline or rejected [9], and for settings with a given time limit where the goal is to maximize the revenue from requests served before the time limit [5]. In contrast to existing work on OLDARP, in this paper we consider requests that need to be served at a specific time that is specified by the request when it is released. Another related problem is the \( k \)-server problem [4, Ch. 10], but in that problem all requests must be served and requests are served at a specific location.

### 1.2 Problem Description and Preliminaries

We consider a setting with only two locations (denoted by 0 and 1) and \( k \) servers (denoted by \( s_1, s_2, \ldots, s_k \)). The \( k \) servers are initially located at location 0. The travel time from 0 to 1 is the same as the travel time from 1 to 0 and is denoted by \( t \).

Let \( R \) denote a sequence of requests that are released over time. The \( i \)-th request is denoted by \( r_i = (\tilde{t}_{r_i}, \tilde{t}_{r_i}, p_{r_i}) \) and is specified by the booking time or release time \( \tilde{t}_{r_i} \), the start time (or pick-up time) \( t_{r_i} \), and the pick-up location \( p_{r_i} \in \{0, 1\} \). If \( r_i \) is accepted, a server must pick up the customer at \( p_{r_i} \) at time \( t_{r_i} \) and drop off the customer at location \( \hat{p}_{r_i} = 1 - p_{r_i} \), the drop-off location of the request, at time \( \tilde{t}_{r_i} = t_{r_i} + t \), the end time (or drop-off time) of the request. We assume that for all \( r_i \in R \), \( t_{r_i} \) is an integer multiple of the travel time between location 0 and location 1, i.e., \( t_{r_i} = vt \) for some \( v \in \mathbb{N} \).

Each server can only serve one request at a time. Scheduling a request yields profit \( r \) \( > 0 \). An empty movement between the two locations takes time \( t \), but has no cost. If two requests are such that they cannot both be served by the same server, we say that the requests are in conflict. We denote the set of requests accepted by an algorithm by \( R' \), and the \( i \)-th request in \( R' \), in order of request start times, is denoted by \( r'_i \). We denote the profit of serving the requests in \( R' \) by \( P_{R'} = r \cdot |R'| \). The goal of the car-sharing problem is to accept a set of requests \( R' \) that maximizes the profit \( P_{R'} \).

The problem for \( k \) servers and two locations for the fixed booking time variant in which \( t_{r_i} - \tilde{t}_{r_i} = a \) for all requests \( r_i \), where \( a \geq t \) is a constant, is called the **kS2L-F** problem. For the variable booking time variant, the booking time \( \tilde{t}_{r_i} \) of any request \( r_i \) must satisfy \( t_{r_i} - b_u \leq \tilde{t}_{r_i} \leq t_{r_i} - b_l \), where \( b_l \) and \( b_u \) are constants, with \( t \leq b_l < b_u \), that specify the minimum and maximum length, respectively, of the time interval between booking time and start time. The problem for \( k \) servers and two locations for the variable booking time variant is called the **kS2L-V** problem. We do not require that the algorithm assigns an accepted request to a server immediately, provided that it ensures that one of the \( k \) servers will serve the request. In our setting, however, it is not necessary for an algorithm to use this flexibility.
An overview of our results is shown in Table 1. In Section 2, we prove the lower bounds. Let \( \nu \) be the number of requests accepted by the adversary and let \( d \) denote the number of requests accepted by the deterministic on-line algorithm. A value \( \rho \) is referred to as \( \rho \)-competitive if \( \frac{\nu}{d} \) is \( \rho \)-competitive and \( \frac{\nu}{d} \) is \( \rho \)-competitive for all request sequences. We use \( \rho \) to denote the objective value produced by an optimal scheduler \( OPT \). The performance of an algorithm for \( kS2L-F \) or \( kS2L-V \) is measured using competitive analysis (see [4]). For any request sequence \( R \), let \( P_{RA} = \sup_{R} \frac{\nu}{d} \). We say that an algorithm \( A \) is \( \rho \)-competitive if \( \frac{\nu}{d} \leq \rho \cdot P_{RA} \) for all request sequences. Let \( ON \) be the set of all on-line algorithms for a problem. We only consider deterministic algorithms. A value \( \beta \) is a lower bound if there is a lower bound \( \beta \) with \( \rho_A \geq \beta \) for all \( A \) in \( ON \). We say that an algorithm \( A \) is optimal if there is a lower bound \( \beta \) with \( \rho_A \geq \beta \).

1.3 Paper Outline

An overview of our results is shown in Table 1. In Section 2, we prove the lower bounds. In Section 3, we propose a balanced greedy algorithm that achieves the best possible competitive ratio. Although variable booking times provide much greater flexibility to customers, we show that our balanced greedy algorithm (only with a different choice of a parameter in the algorithm) is still optimal. When \( k \neq 3i \) (resp. \( k \neq 5i \), \( i \in \mathbb{N} \)), the upper bounds for \( kS2L-V \) when \( b_t \geq t \) and \( b_u - b_t < t \) (resp. \( b_u - b_t \geq t \)) are only slightly worse. The proofs for the latter cases are omitted due to space restrictions.

2 Lower Bounds

In this section, we present lower bounds for \( kS2L-F \) and \( kS2L-V \). We use \( ALG \) to denote any deterministic on-line algorithm and \( OPT \) to denote an optimal scheduler. The set of requests accepted by \( ALG \) is referred to as \( R' \), and the set of requests accepted by \( OPT \) is referred to as \( R^* \).

**Theorem 1.** For \( a \geq t \) (resp. \( b_t \geq t, b_u - b_t < t \)), no deterministic on-line algorithm for \( kS2L-F \) (resp. \( kS2L-V \)) can achieve competitive ratio smaller than 1.5.

**Proof.** Initially, the adversary releases the 1st request sequence \( r_1, r_2, \ldots, r_k \) with \( r_1 = r_2 = \cdots = r_k = (\nu \cdot t - a, \nu \cdot t, 1) \), where \( \nu \in \mathbb{N} \) and \( \nu \cdot t - a \geq 0 \) (resp. \( r_1 = r_2 = \cdots = r_k = (\nu \cdot t - b_t, \nu \cdot t, 1) \) where \( \nu \in \mathbb{N} \) and \( \nu \cdot t - b_t \geq 0 \)). Suppose \( ALG \) accepts \( k_1 \) (\( 1 \leq k_1 \leq k \)) requests in the 1st request sequence. There are two options that the adversary can adopt:

**Option 1:** The adversary releases the 2nd request sequence \( r_{k+1}, r_{k+2}, \ldots, r_{2k} \) with \( r_{k+1} = r_{k+2} = \cdots = r_{2k} = (r_{t_1}, t_{r_1}, 0) \), and the 3rd request sequence \( r_{2k+1}, r_{2k+2}, \ldots, r_{3k} \) with \( r_{2k+1} = r_{2k+2} = \cdots = r_{3k} = (t_{r_1} + t, t_{r_1} + t_1, 1) \). Note that the requests in the 2nd and the 3rd request sequences must be assigned to different servers from the \( k_1 \) servers that have accepted requests of the 1st request sequence as they are in conflict. From this it follows that \( ALG \) cannot accept more than \( 2(k - k_1) \) requests of the 2nd and the 3rd request sequences. \( OPT \) accepts all the requests in the 2nd and the 3rd request sequences. We have \( P_{RA} = 2kr \) and \( P_{RA} \leq k_1 r + 2(k - k_1) r = (2k - k_1) r \), and hence \( \frac{\nu}{d} \geq \frac{2k - k_1}{k_1} \).

**Option 2:** The adversary does not release any more requests. \( OPT \) accepts all requests in the 1st request sequence. We have \( P_{RA} = k \cdot r \) and \( P_{RA} = k_1 \cdot r \), and hence \( \frac{\nu}{d} \geq \frac{k}{k_1} \).
Algorithm 1 Balanced Greedy Algorithm (BGA).

Input: $k$ servers ($2\theta k$ specified servers and $(1 - 2\theta) k$ unspecified servers), requests arrive over time.

Step: When request $r_i$ arrives, if it is acceptable to a specified server, assign it to that server; otherwise, if $r_i$ is acceptable to an unspecified server, assign $r_i$ to that server; otherwise, reject it.

If $k_1 \geq \frac{2\theta}{1 + \frac{2\theta - \theta}{2\theta - 1}} \geq 1.5$; if $k_1 \leq \frac{2\theta}{1 + \frac{2\theta - \theta}{2\theta - 1}} \geq 1.5$. As the adversary can choose the option that maximizes $\frac{P_{\text{opt}}}{P_{\text{ALG}}}$, the claimed lower bound of 1.5 follows.

Theorem 2. For $b_t \geq t$ and $b_u - b_t \geq t$, no deterministic on-line algorithm for kS2L-V can achieve competitive ratio smaller than $5/3$.

Proof. Initially, the adversary releases the 1st request sequence $r_1, r_2, \ldots, r_k$ with $r_1 = r_2 = \cdots = r_k = (\nu \cdot t - b_u, \nu \cdot t, 0)$ (where $\nu \in \mathbb{N}$ with $\nu \cdot t - b_u \geq 0$). Suppose ALG accepts $k_1$ ($1 \leq k_1 \leq k$) requests in the 1st request sequence. There are now two options that the adversary can adopt.

Option 1: The adversary releases the 2nd request sequence $r_{k+1}, r_{k+2}, \ldots, r_{2k}$ with $r_{k+1} = r_{k+2} = \cdots = r_{2k} = (\tilde{t}_r, t - t, 0)$ (note that $t - \tilde{t}_r = \nu \cdot t - t - (\nu \cdot t - b_u) = b_u - t \geq b_1$), and the 3rd request sequence $r_{2k+1}, r_{2k+2}, \ldots, r_{3k}$ with $r_{2k+1} = r_{2k+2} = \cdots = r_{3k} = (\tilde{t}_r + t, t - t, 0)$, and the 4th request sequence $r_{3k+1}, r_{3k+2}, \ldots, r_{4k}$ with $r_{3k+1} = r_{3k+2} = \cdots = r_{4k} = (\tilde{t}_r + 2t, t - t, 0)$.

Note that the requests in the 2nd, the 3rd and the 4th request sequences must be assigned to different servers from the $k_1$ servers that have accepted requests of the 1st request sequence as they are in conflict. From this it follows that ALG cannot accept more than $3(k - k_1)$ requests in the 2nd, 3rd and 4th request sequences. OPT accepts all the requests in the 2nd, 3rd and 4th request sequences. We have $P_{\text{opt}} = 3kr$ and $P_{\text{ALG}} \leq k_1 r + 3(k - k_1) r = (3k - 2k_1) r$, and hence $\frac{P_{\text{opt}}}{P_{\text{ALG}}} \geq \frac{3k}{2k - 2k_1}$.

Option 2: The adversary does not release any more requests. OPT accepts all requests in the 1st request sequence. We have $P_{\text{opt}} = k \cdot r$ and $P_{\text{ALG}} = k_1 \cdot r$, and hence $\frac{P_{\text{opt}}}{P_{\text{ALG}}} \geq \frac{k}{k_1}$.

If $k_1 \geq \frac{3}{2} k$, $\frac{3k}{3k - 2k_1} \geq \frac{5}{4}$; if $k_1 \leq \frac{3}{2} k$, $\frac{k}{k_1} \geq \frac{5}{4}$. As the adversary can choose the option that maximizes $\frac{P_{\text{opt}}}{P_{\text{ALG}}}$, the claimed lower bound of $5/3$ follows.

3 Upper Bounds

We propose a Balanced Greedy Algorithm (BGA) for the kS2L-F/V problem, shown in Algorithm 1. The $k$ servers are divided into two groups: a set $S_f$ of $2\theta k$ specified servers and a set $S_u$ of $(1 - 2\theta) k$ unspecified servers, where $\theta$ is a parameter satisfying $0 \leq \theta \leq \frac{1}{2}$ and chosen in such a way that $\theta k$ is an integer. The set $S_f$ is further partitioned into sets $S_f^o$ and $S_f^s$ of $\theta k$ servers each. The $\theta k$ specified servers in $S_f^o$ serve requests that start at location 0 at time $\nu t$ where $\nu$ is even and requests that start at location 1 at time $\nu t$ where $\nu$ is odd. The $\theta k$ specified servers in $S_f^s$ serve the other request types, i.e., requests that start at location 0 (resp. 1) at time $\nu t$ where $\nu$ is odd (resp. even).

When the algorithm receives request $r_i$, let $R'(r_i)$ denote the set of requests that BGA has already accepted, and let $R'_i(r_i)$ denote the set of requests that BGA has accepted and that are assigned to $s_j$, for any $j$. Request $r_i$ is acceptable to a specified server if and only if the number of requests in $R'(r_i)$ that start at $t_x$ and have pick-up location $p_x$ is less than
θk. Furthermore, ri is acceptable to an unspecified server sj (sj ∈ Su) if and only if ri is not in conflict with the requests in R′(ri), i.e., for all r′ j ∈ R′(ri) we have |t ri − t r′ j | ≥ 2t if pr = p r′ j and |t ri − t r′ j | ≥ t if pr ≠ p r′ j.

Denote the requests accepted by OPT by \( R^* = \{ r^*_1, r^*_2, \ldots, r^*_{|R^*|} \} \) and the requests accepted by BGA by \( R' = \{ r'_1, r'_2, \ldots, r'_{|R'|} \} \) indexed in order of non-decreasing start times. The requests with equal start time are ordered in the order in which they arrive. Let \( R^*(d) \) denote the set of requests in \( R^* \) which start at time \( d \), and let \( R^*(d, e) \) denote the set of requests in \( R^* \) which start at time \( d \) and have pick-up location \( e \). Observe that for all \( d, e \), we have \( |R^*(d)| \leq k \) and \( |R^*(d, e)| \leq k \). Let \( R'(d) \) denote the set of requests in \( R' \) which start at time \( d \), and let \( R'(d, e) \) denote the set of requests in \( R' \) which start at time \( d \) and have pick-up location \( e \). Observe that for all \( d, e \), we have \( |R'(d)| \leq k \) and \( |R'(d, e)| \leq (1 - \theta)k \).

For simplification of the analysis, we suppose that the specified servers in each of the sets \( S^*_1 \) and \( S^*_2 \) are ordered and if a request \( r_i \) is acceptable to some specified server, BGA assigns \( r_i \) to the available specified server that comes first in that order.

> **Observation 3.** If \( \theta > 0 \), then \( \forall r^*_i \in R^*: t^*_r \leq t^*_i \leq t^*_{r^*_i} \).

> **Observation 4.** For every \( r^*_i \in R^* \), BGA accepts \( \min \{|R^*(t^*_i, p^*_i)|, \theta k\} \) requests that start at \( t^*_i \) and have pick-up location \( p^*_i \) with specified servers, and hence \( |R'(t^*_i, p^*_i)| \geq \min \{|R^*(t^*_i, p^*_i)|, \theta k\} \).

> **Observation 5.** If \( k_0 \) servers of OPT, where \( 0 \leq k_0 \leq k \), each accept \( y \) \((y \geq 1) \) requests that start during period \( [x, x + yt] \) (where \( x = vt \) for some \( v \in \mathbb{N} \)), then at least \( \min \{\theta k, k_0\} \) specified servers of BGA each accept \( y \) requests that start during this period.

To illustrate the idea of our analysis, we first give a simple proof of an upper bound of 2 on the competitive ratio of BGA.

> **Theorem 6.** With \( \theta = \frac{1}{2} \), BGA is 2-competitive for kS2L-F and kS2L-V if \( k \) is even.

**Proof.** Since BGA with \( \theta = \frac{1}{2} \) accepts a request \( r_i \in R \) if the number of requests in \( R'(r_i) \) that start at \( t_i \) and have pick-up location \( p_i \) is less than \( \frac{k}{2} \), BGA can always accept \( \min \{\frac{k}{2}, |R'(t_i, p_i)|\} \) requests that start at the same time and have the same pick-up location. As OPT accepts at most \( k \) requests that start at the same time and have the same pick-up location, i.e., \( |R^*(t^*_i, p^*_i)| \leq k \), it follows that \( |R'| \geq \frac{1}{2} |R^*| \).

> **Definition 7 (Common and uncommon request).** For each \( r^*_i \in R^* \), if the number of requests in \( R^* \) that start at \( t^*_i \) and have pick-up location \( p^*_i \) is no more than the number of requests in \( R^* \) which start at \( t^*_i \) and have pick-up location \( p^*_i \), i.e., \( |R^*(t^*_i, p^*_i)| \leq |R^*(t^*_i, p^*_i)| \), we say that the requests in \( R^*(t^*_i, p^*_i) \) are common; if the number of requests in \( R^* \) which start at \( t^*_i \) and have pick-up location \( p^*_i \) is greater than the number of requests in \( R^* \) which start at \( t^*_i \) and have pick-up location \( p^*_i \), i.e., \( |R^*(t^*_i, p^*_i)| > |R^*(t^*_i, p^*_i)| \), we say that the first \( |R^*(t^*_i, p^*_i)| \) requests in \( R^*(t^*_i, p^*_i) \) are common, and the remaining requests in \( R^*(t^*_i, p^*_i) \) are uncommon.

> **Observation 8.** If \( r^*_i \in R^* \) is uncommon, \( |R'(t^*_i, p^*_i)| \geq \theta k \).

> **Definition 9 (Sufficient and insufficient interval).** We say that an interval \([x, x + t] \) (\( x \) is an integer multiple of \( t \)) is sufficient if \( |R'(x)| \geq (1 - \theta)|R^*(x)| \); otherwise it is insufficient.
3.1 Upper Bounds for kS2L-F

- **Observation 10.** For kS2L-F, if interval $[x, x+t]$ ($x$ is an integer multiple of $t$) is insufficient, then $x \geq t_r + t$.

With the following two lemmas, we show that if an interval $I$ is insufficient, the interval $I'$ preceding it must be sufficient and the competitive ratio of BGA with respect to requests starting in $I$ and $I'$ is at most 1.5 (for $\theta = \frac{1}{3}$).

- **Lemma 11.** For $\frac{1}{3} \leq \theta \leq \frac{1}{2}$, if $r_*^t \in R^*$ is uncommon and interval $[t_r^*, t_r^* + t)$ is insufficient, then $|R'(t_r^* - t, p_v^r)| = k - |R'(t_r^*, p_v^r)|$.

**Proof.** As $r_*^t$ is uncommon, $|R'(t_r^*, p_v^r)| \geq \theta k$ (by Observation 8) and every unspecified server has accepted a request that is in conflict with $r_*^t$, i.e., for all $s_j \in S_u$, there is $r_j^t \in R'_j(r_*^t)$ (recall that $R'_j(r_*^t)$ is the set of requests that are accepted and assigned to $s_j$ by BGA at the time when $r_*^t$ is released) such that $t_r^* = t_q$ and $p_v^r = p_v^{'t}$, or $t_r^* - t_q = t$ and $p_v^r = p_v^{'t}$ or $t_r^* = t_q$ and $p_v^r \neq p_v^{'t}$.

Observe that $|R'(t_r^* - t, p_v^r)| < (1-\theta)k$ because interval $[t_r^*, t_r^* + t)$ is insufficient. Since $|R'(t_r^*, p_v^r)| \geq \theta k$, $|R'(t_r^*, p_v^r)| < (1-\theta)k - \theta k \leq \theta k$ (as $\frac{1}{3} \leq \theta \leq \frac{1}{2}$). This implies that BGA does not use unspecified servers to serve requests in $R'(t_r^*, p_v^r)$ because BGA does not use unspecified servers when specified servers are available. From this it follows that each of the unspecified servers either accepts a request that starts at $t_r^*$ with pick-up location $p_v^r$, or accepts a request that starts at $t_r^* - t$ with pick-up location $p_v^r$. As $|R'(t_r^*, p_v^r)| < (1-\theta)k = \theta k + (1-2\theta)k$, at least one unspecified server accepts a request that starts at $t_r^* - t$ with pick-up location $p_v^r$. This implies that $|R'(t_r^* - t, p_v^l)| \geq \theta k$. Since $|R'(t_r^*, p_v^l)| \geq \theta k$, each of the specified servers either accepts a request that starts at $t_r^*$ at $p_v^r$ or a request that starts at $t_r^* - t$ at $p_v^r$. Therefore $|R'(t_r^* - t, p_v^r)| = k - |R'(t_r^*, p_v^r)|$.

- **Lemma 12.** For $\theta = \frac{1}{3}$, if $r_*^t \in R^*$ is uncommon and interval $[t_r^*, t_r^* + t)$ is insufficient, then $|R'(t_r^* - t) + |R'(t_r^*)| \geq \frac{2}{3}(|R'(t_r^* - t)| + |R'(t_r^*)|)$ and also $|R'(t_r^* - t)| > \frac{2}{3}|R'(t_r^* - t)|$.

**Proof.** According to Lemma 11, $|R'(t_r^* - t, p_v^r)| = k - |R'(t_r^*, p_v^r)|$. From this it follows that each server of BGA accepts at least one request that starts during period $[t_r^* - t, t_r^*]$, i.e., $|R'(t_r^* - t)| + |R'(t_r^*)| \geq k$. Suppose $k_0$ servers of $OPT$ each accept two requests that start during period $[t_r^* - t, t_r^*]$. We distinguish two cases.

**Case 1:** $k_0 \geq \frac{k}{3}$. By Observation 5 (with $y = 2$), at least $\theta k$ servers of BGA accept two requests that start during period $[t_r^* - t, t_r^*]$. Since each server accepts at least one request that starts during period $[t_r^* - t, t_r^*]$, $|R'(t_r^* - t)| + |R'(t_r^*)| \geq 2\theta k + (1-\theta)k = \frac{4}{3}k$.

Since $|R'(t_r^* - t)| + |R'(t_r^*)| \leq 2k$ (each server of $OPT$ accepts at most two requests that start during period $[t_r^* - t, t_r^*]$), we have $\frac{|R'(t_r^* - t)| + |R'(t_r^*)|}{|R'(t_r^* - t)| + |R'(t_r^*)|} \leq \frac{2k}{2k} = \frac{2}{3}$.

**Case 2:** $k_0 < \frac{k}{3}$. Note that each server of $OPT$ accepts at most two requests that start during period $[t_r^* - t, t_r^*]$, so $|R'(t_r^* - t)| + |R'(t_r^*)| < \frac{2}{3}k + (k - \frac{2}{3}k) = \frac{4}{3}k$. Since $|R'(t_r^* - t)| + |R'(t_r^*)| \geq k$, we have $\frac{|R'(t_r^* - t)| + |R'(t_r^*)|}{|R'(t_r^* - t)| + |R'(t_r^*)|} \leq \frac{4k}{k} = \frac{4k}{k} < \frac{3}{2}$.

Because $|R'(t_r^*)| < \frac{2}{3}|R'(t_r^*)|$ and $|R'(t_r^* - t)| + |R'(t_r^*)| \geq \frac{2}{3}(|R'(t_r^* - t)| + |R'(t_r^*)|)$, we have $|R'(t_r^* - t)| > \frac{3}{2}|R'(t_r^* - t)|$.

- **Corollary 13.** If interval $[x, x + t]$ ($x$ is an integer multiple of $t$) is insufficient, then interval $[x - t, x]$ and interval $[x + t, x + 2t]$ are sufficient.
Algorithm 2 Partition Rule (for kS2L-F).

Initialization: \( \gamma = \frac{t_r' - t_r'}{\nu t}, j = 2, l_j = 0, i = 0. \)
while \( i \leq \gamma \) do
  if interval \( i \) and \( i + 1 \) are sufficient then
    
  else if interval \( i \) is sufficient and interval \( i + 1 \) is insufficient then
    
\( \gamma' = j. \)

\[ \text{Theorem 14.} \] With \( \theta = \frac{1}{3} \), BGA is \( \frac{3}{2} \)-competitive for kS2L-F if \( k = 3\nu \) (\( \nu \in \mathbb{N} \)).

\[ \text{Proof.} \] We partition the time horizon \([0, \infty)\) into \( \gamma' \) (\( \gamma' \leq \gamma + 3 \), where \( \gamma = \frac{t_r' - t_r'}{\nu t} \)) periods that can be analyzed independently. Let interval \( i \) (\( 0 \leq i \leq \gamma \)) denote interval \([t_r' + it, t_r' + (i+1)t]\). We partition the time horizon based on the Partition rule (Algorithm 2) and let period \( j \) (\( 1 \leq j < \gamma' \)) denote \([t_r' + l_j \cdot t, t_r' + l_j + 1 \cdot t]\), in such a way that BGA and OPT do not accept any requests in the first period \([0, t_r')\) and the last period \([t_r' + t, \infty)\), and the length of each period \( j \) (\( 1 \leq j < \gamma' \)), i.e., \((l_{j+1} - l_j) t\), is either \( t \) or \( 2t \). By Corollary 13 and the Partition rule (Algorithm 2), we have the following properties: if the length of period \( j \) is \( t \), i.e., \( l_{j+1} - l_j = 1 \), period \( j \) is sufficient; if the length of period \( j \) is \( 2t \), i.e., \( l_{j+1} - l_j = 2 \), the first half of period \( j \), i.e., \([t_r' + l_j \cdot t, t_r' + (l_j + 1) \cdot t]\), is sufficient and the second half of period \( j \), i.e., \([t_r' + (l_j + 1) \cdot t, t_r' + (l_j + 2) \cdot t]\), is insufficient. Recall that interval 0 is always sufficient by Observation 10.

Let \( R_{(j)} \) denote the set of requests accepted by OPT that start in period \( j \), for \( 1 \leq j \leq \gamma' \). Let \( R_{(j)}' \) denote the set of requests accepted by BGA that start in period \( j \), for \( 1 \leq j \leq \gamma' \). We bound the competitive ratio of BGA by analyzing each period independently. As \( R' = \bigcup_j R_{(j)}' \) and \( R = \bigcup_j R_{(j)} \), it is clear for any \( \alpha \geq 1 \) that \( P_{R'}/P_R \leq \alpha \) if we can show that \( P_{R_{(j)}'}/P_{R_{(j)}} \leq \alpha \) for all \( j, 1 \leq j \leq \gamma' \).

According to Lemma 12, when \( l_{j+1} - l_j = 2 \), \( P_{R_{(j)}'}/P_{R_{(j)}} \leq \frac{3}{2} \). Based on the partition rule, when \( l_{j+1} - l_j = 1 \), period \( j \) is sufficient, i.e., \([R_{(j)}'] \geq (1 - \theta)|R_{(j)}|\) and hence \( P_{R_{(j)}'}/P_{R_{(j)}} \leq \frac{3}{2} \). Since \( P_{R_{(j)}'} = P_{R_{(j)}} = 0 \) for \( j = 1 \) and \( j = \gamma' \) (recall that by Observation 3, all requests accepted by BGA and OPT do not start earlier than \( t_r' \) and do not start later than \( t_r' + t \)), the theorem follows.

3.2 Upper Bounds for kS2L-V

If \( b_t \geq t \) and \( b_u - b_t < t \) for the kS2L-V problem, let \( \theta = \frac{1}{3} \). Since each request starts at time \( \nu t \) for some \( \nu \in \mathbb{N} \), all requests start in order of their release times, and therefore the upper bound for the kS2L-V problem is equal to the upper bound for the kS2L-F problem (with \( a \geq t \)). From now on consider the kS2L-V problem with \( b_t \geq t \) and \( b_u - b_t \geq t \), and let \( \theta = \frac{2}{3} \).

\[ \text{Lemma 15.} \] For \( \theta = \frac{2}{3} \), if \( r_{(j)}' \in R' \) is uncommon and interval \([t_r', t_r' + t]\) is insufficient, then one of the following holds:

(i) \(|R'(t_r' - t, p_{r})| = k - |R'(t_r', p_{r})| > \frac{2}{3}k| \) and \(|R'(t_r', p_{r})| \leq \theta k, or \)

(ii) \(|R'(t_r' + t, p_{r})| = k - |R'(t_r', p_{r})| > \frac{2}{3}k| \) and \(|R'(t_r', p_{r})| \leq \theta k, or \)

(iii) \(|R'(t_r' - t, p_{r})| > \theta k \) and \(|R'(t_r', p_{r})| \geq (1 - 2\theta)k \) and \(|R'(t_r' + t, p_{r})| > \theta k \).
Proof. As \( r^*_t \) is uncommon, \(|R'(t^*_t, p^*_t)| \geq \theta k\) (by Observation 8) and every unspecified server has accepted a request that is in conflict with \( r^*_t \), i.e., for every \( s_j \in S_u \) there exists \( r^*_j \in R'(r^*_t)\) (recall that \( R'(r^*_t) \) is the set of requests that are accepted and assigned to \( s_j \) by BGA at the time \( r^*_t \) arrives) such that \( t^*_j = t^*_t \) and \( p^*_j = p^*_t \), or \( t^*_j = t^*_t - t \) and \( p^*_j = p^*_t \), or \( t^*_j = t^*_t + t \) and \( p^*_j = p^*_t \), or \( t^*_j = t^*_t - t \) and \( p^*_j \neq p^*_t \).

Observe that \(|R'(t^*_t)| < (1 - \theta)k\) because interval \([t^*_t, t^*_t + t]\) is insufficient. Since \(|R'(t^*_t, p^*_t)| \geq \theta k\), \(|R'(t^*_t, p^*_t)| < (1 - \theta)k - \theta k \leq \theta k\) (as \( \theta = \frac{\delta}{2} \)). This implies that BGA does not use unspecified servers to serve requests in \( R'(t^*_t, p^*_t)\) because BGA does not use unspecified servers when specified servers are available. From this it follows that each of the unspecified servers either accepts a request that starts at \( t^*_t \) with pick-up location \( p^*_t \), or accepts a request that starts at \( t^*_t - t \) with pick-up location \( p^*_t \), or accepts a request that starts at \( t^*_t + t \) with pick-up location \( p^*_t \). As \(|R'(t^*_t, p^*_t)| < (1 - \theta)k = \theta k + (1 - 2\theta)k\), at least one unspecified server accepts a request that starts at \( t^*_t - t \) at \( p^*_t \), or a request that starts at \( t^*_t + t \) at \( p^*_t \). We distinguish three cases.

Case 1: No unspecified server accepts a request that starts at \( t^*_t + t \) at \( p^*_t \). Then at least one unspecified server accepts a request that starts at \( t^*_t + t \) at \( p^*_t \). This implies that \(|R'(t^*_t + t, p^*_t)| \geq \theta k\). Since \(|R'(t^*_t, p^*_t)| \geq \theta k\), each of the specified servers either accepts a request that starts at \( t^*_t \) at \( p^*_t \) or a request that starts at \( t^*_t - t \) at \( p^*_t \). Each of the unspecified servers either accepts a request that starts at \( t^*_t \) at \( p^*_t \), or a request that starts at \( t^*_t - t \) at \( p^*_t \). Therefore, \(|R'(t^*_t - t, p^*_t)| = k - |R'(t^*_t, p^*_t)|\), and (i) holds.

Case 2: No unspecified server accepts a request that starts at \( t^*_t - t \) at \( p^*_t \). By symmetric arguments to Case 1, we get \(|R'(t^*_t + t, p^*_t)| = k - |R'(t^*_t, p^*_t)|\), and (ii) holds.

Case 3: At least one unspecified server accepts a request that starts at \( t^*_t + t \) at \( p^*_t \), and at least one unspecified server accepts a request that starts at \( t^*_t - t \) at \( p^*_t \). This implies that \(|R'(t^*_t - t, p^*_t)| \geq \theta k\) and \(|R'(t^*_t + t, p^*_t)| \geq \theta k\). Since each of the unspecified servers either accepts a request that starts at \( t^*_t \) at \( p^*_t \), or a request that starts at \( t^*_t - t \) at \( p^*_t \), or a request that starts at \( t^*_t + t \) at \( p^*_t \), we have that \(|R'(t^*_t - t, p^*_t)| = \theta k + |R'(t^*_t + t, p^*_t)| \geq \theta k \geq (1 - 2\theta)k\), and (iii) holds.

Definition 16 (l-full and r-full, l-large and r-large, l-small and r-small). If \( r^*_t \) is an uncommon request such that the interval \([t^*_t - t, t^*_t + t]\) is insufficient, we say that the interval \([t^*_t - t, t^*_t + t]\) (resp. \([t^*_t + t, t^*_t + 2t]\)) is l-full (resp. r-full) with respect to \( I \) if \(|R'(t^*_t - t, p^*_t)| = k - |R'(t^*_t, p^*_t)|\) (resp. \(|R'(t^*_t + t, p^*_t)| = k - |R'(t^*_t, p^*_t)|\)); we say that the interval \([t^*_t - t, t^*_t + t]\) (resp. \([t^*_t + t, t^*_t + 2t]\)) is l-large (resp. r-large) with respect to \( I \) if \( \frac{\delta}{2}k < |R'(t^*_t - t, p^*_t)| < k - |R'(t^*_t, p^*_t)|\) (resp. \( \frac{\delta}{2}k < |R'(t^*_t + t, p^*_t)| < k - |R'(t^*_t, p^*_t)|\)); and we say that the interval \([t^*_t - t, t^*_t]\) (resp. \([t^*_t + t, t^*_t + 2t]\)) is l-small (resp. r-small) with respect to \( I \) if \(|R'(t^*_t - t, p^*_t)| \leq \frac{\delta}{2}k\) (resp. \(|R'(t^*_t + t, p^*_t)| \leq \frac{\delta}{2}k\)).

Note that the properties l-full, l-large and l-small refer to the interval directly to the left of an insufficient interval, and the properties r-full, r-large and r-small to the interval directly to the right of an insufficient interval.

Observation 17 (Uniqueness). If \( r^*_t \) is uncommon and interval \([t^*_t - t, t^*_t + t]\) is insufficient, then interval \([t^*_t - t, t^*_t]\) is either l-full, l-large, or l-small, and interval \([t^*_t - t, t^*_t + 2t]\) is either r-full, r-large, or r-small.

By Lemma 15, we obtain:

Observation 18. If \( r^*_t \) is uncommon, interval \([t^*_t, t^*_t + t]\) is insufficient, and interval \([t^*_t + t, t^*_t + 2t]\) is r-small, then interval \([t^*_t, t^*_t + t]\) is l-full. Similarly, if \( r^*_t \) is uncommon, interval \([t^*_t, t^*_t + t]\) is insufficient, and interval \([t^*_t - t, t^*_t]\) is l-small, then interval \([t^*_t, t^*_t + t]\) is r-full.
Lemma 19. For  $\theta = \frac{3}{2}k$, if $r^*_i \in R^*$ is uncommon, interval $[t^*_r, t^*_r + t]$ is insufficient and $|R'(t^*_r + t, p^*_i)| > \frac{\theta}{2}k$ (i.e., interval $[t^*_r, t^*_r + t]$ is $k$-large or $k$-full), then $|R'(t^*_r + t) + R'(t^*_r)| \geq \frac{\theta}{2}(|R'(t^*_r + t)| + |R'(t^*_r)|)$ and interval $[t^*_r, t^*_r + t]$ is sufficient. Similarly, if $r^*_i$ is uncommon, interval $[t^*_r, t^*_r + t]$ is insufficient and $|R'(t^*_r + t, p^*_i)| > \frac{\theta}{2}k$ (i.e., interval $[t^*_r + t, t^*_r + 2t]$ is $r$-large or $r$-full), then $|R'(t^*_r + t) + R'(t^*_r)| \geq \frac{\theta}{2}(|R'(t^*_r + t)| + |R'(t^*_r)|)$ and interval $[t^*_r + t, t^*_r + 2t]$ is sufficient.

Proof. Observe that $|R'(t^*_r, p^*_i)| \geq \frac{\theta}{2}k$ because $r^*_i$ is uncommon. Since $|R'(t^*_r - t, p^*_i)| \geq \frac{\theta}{2}k$, each specified server of BGA accepts at least one request that starts during period $[t^*_r - t, t^*_r]$ (resp. $[t^*_r, t^*_r + t]$). Suppose $k_0$ servers of OPT each accept two requests that start during period $[t^*_r - t, t^*_r]$ (resp. $[t^*_r, t^*_r + t]$). We distinguish two cases.

Case 1: $k_0 \geq \frac{\theta}{2}k$. By Observation 5 (with $y = 2$), at least $\frac{\theta}{2}k$ servers of BGA each accept two requests that start during period $[t^*_r - t, t^*_r]$ (resp. $[t^*_r, t^*_r + t]$).

Since each specified server of BGA accepts at least one request that starts during period $[t^*_r - t, t^*_r]$ (resp. $[t^*_r, t^*_r + t]$), $|R'(t^*_r - t) + R'(t^*_r)| \geq \frac{\theta}{2}k + \frac{\theta}{2}k = \frac{\theta}{2}k$ (resp. $|R'(t^*_r + t) + R'(t^*_r)| \geq \frac{\theta}{2}k$).

Since $|R'(t^*_r - t) + R'(t^*_r)| \leq 2k$ and $|R'(t^*_r + t) + R'(t^*_r)| \leq 2k$ (each server of OPT accepts at most two requests that start during period $[t^*_r - t, t^*_r]$ or period $[t^*_r + t, t^*_r + 2t]$), we have $\frac{|R'(t^*_r - t) + R'(t^*_r)|}{|R'(t^*_r + t) + R'(t^*_r)|} \leq \frac{2k}{\frac{\theta}{2}k} = \frac{\theta}{k}$ (resp. $\frac{|R'(t^*_r + t) + R'(t^*_r)|}{|R'(t^*_r - t) + R'(t^*_r)|} \leq \frac{\theta}{k}$).

Observe that $|R'(t^*_r)| < \frac{\theta}{2}k$ because interval $[t^*_r, t^*_r + t]$ is insufficient. If $|R'(t^*_r - t, p^*_i)| > \frac{\theta}{2}k$, then $|R'(t^*_r - t) + R'(t^*_r)| \geq \frac{\theta}{2}(|R'(t^*_r - t)| + |R'(t^*_r)|)$ implies $|R'(t^*_r - t) \geq \frac{\theta}{2}|R'(t^*_r)|$. Similarly, if $|R'(t^*_r + t, p^*_i)| > \frac{\theta}{2}k$, then $|R'(t^*_r + t)| \geq \frac{\theta}{2}|R'(t^*_r)|$.

Lemma 20. For $\theta = \frac{3}{2}k$, consider any $r^*_i, r^*_j \in R^*$ where $t^*_r = t^*_r + 2t, t^*_r$ and $r^*_j$ are uncommon, intervals $[t^*_r, t^*_r + t]$ and $[t^*_r, t^*_r + t]$ are insufficient, and interval $[t^*_r + t, t^*_r + 2t]$ is $r$-full (i.e., $|R'(t^*_r + t, p^*_i)| = k - |R'(t^*_r, p^*_i)|$). Then $|R'(t^*_r)| + |R'(t^*_r + t)| + |R'(t^*_r + 2t)| \geq \frac{\theta}{2}(|R'(t^*_r)| + |R'(t^*_r + t)| + |R'(t^*_r + 2t)|).

Proof. Observe that $|R'(t^*_r, p^*_i)| \geq \frac{\theta}{2}k$ (as $r^*_i$ is uncommon), $|R'(t^*_r + t, p^*_i)| \geq \frac{\theta}{2}k$ (as $r^*_j$ is uncommon). From this it follows that at least $\frac{\theta}{2}k$ specified servers of BGA each at least accept two requests that start during period $[t^*_r, t^*_r]$. Since $|R'(t^*_r, p^*_i)| = 1 - |R'(t^*_r, p^*_i)|$, each server of BGA at least accepts one request that starts during period $[t^*_r, t^*_r]$. Suppose $k_0$ servers of OPT each accept three requests that start during period $[t^*_r, t^*_r]$. We distinguish two cases.

Case 1: $k_0 \geq \frac{\theta}{2}k$. By Observation 5 (with $y = 3$), at least $\frac{\theta}{2}k$ servers of BGA each accept three requests that start during period $[t^*_r, t^*_r]$. Since each server of BGA accepts at least one request that starts during period $[t^*_r, t^*_r]$, $|R'(t^*_r) + |R'(t^*_r + t)| + |R'(t^*_r + 2t)| \geq \frac{\theta}{2}(|R'(t^*_r)| + |R'(t^*_r + t)| + |R'(t^*_r + 2t)|).$
Algorithm 3 Partition Rule (for kS2L-V).

Initialization: \( \gamma = \frac{t_{r_1} - t_{r_i}}{m} \), \( j = 2 \), \( l_j = 0 \), \( i = 0 \).

while \( i \leq \gamma \) do

if interval \( i \) and interval \( i + 1 \) are Suf, then

\( j = j + 1 \), \( i = i + 1 \), \( l_j = i \);

else if interval \( i \) is Suf and l-small, and interval \( i + 1 \) is InSuf, then

\( j = j + 1 \), \( i = i + 1 \), \( l_j = i \);

else if interval \( i \) is Suf and not l-small, and interval \( i + 1 \) is InSuf, then

\( j = j + 1 \), \( i = i + 2 \), \( l_j = i \);

else if interval \( i \) is InSuf, interval \( i + 1 \) is r-full and interval \( i + 2 \) is InSuf, then

\( j = j + 1 \), \( i = i + 3 \), \( l_j = i \);

else if interval \( i \) is InSuf, interval \( i + 1 \) is r-full and interval \( i + 2 \) is Suf, then

\( j = j + 1 \), \( i = i + 2 \), \( l_j = i \);

\( \gamma' = j \).

Case 2: \( k_0 < \frac{2}{7}k \). By Observation 5 (with \( y = 3 \)), at least \( k_0 \) servers of BGA each accept three requests that start during period \([t_{r_1}, t_{r_2}]\). Since each server of BGA accepts at least one request that starts during period \([t_{r_1}, t_{r_2}]\) and at least \( \frac{2}{7}k \) specified servers of BGA each accept at least two requests that start during period \([t_{r_1}, t_{r_2}], [R'(t_{r_2})] + [R(t_{r_2} + t)] + [R(t_{r_2} + 2t)] \geq 3k_0 + 2 \cdot \left( \frac{2}{7}k - k_0 \right) + \frac{3}{7}k \geq \frac{7}{5}k + k_0 \). Since \( |R'(t_{r_2})| + |R(t_{r_2} + t)| + |R(t_{r_2} + 2t)| \geq 3k_0 + 2(k - k_0) = 2k + k_0 \), we have \( \frac{|R'(t_{r_2})| + |R(t_{r_2} + t)| + |R(t_{r_2} + 2t)|}{|R'(t_{r_2})| + |R(t_{r_2} + t)| + |R(t_{r_2} + 2t)|} \leq \frac{2k + k_0}{\frac{7}{5}k + k_0} \leq \frac{10}{7} < \frac{5}{3} \).

\( \blacktriangleright \) Theorem 21. With \( \theta = \frac{2}{5} \), BGA is \( \frac{5}{3} \)-competitive for kS2L-V if \( k = 5\nu \) (\( \nu \in \mathbb{N} \)).

Proof. We partition the time horizon \([0, \infty)\) into \( \gamma' \) (\( \gamma' = \gamma + 3, \gamma = \frac{t_{r_1} - t_{r_2}}{m} \)) periods that can be analyzed independently. Let interval \( i \) (\( 0 \leq i \leq \gamma \)) denote the interval \([t_{r_1} + it, t_{r_2} + (i + 1)t]\). We partition the time horizon using the partition rule shown in Algorithm 3, where we use InSuf as an abbreviation for insufficient and Suf as an abbreviation for sufficient. We let period \( j \) (\( 1 \leq j \leq \gamma' \)) denote \([t_{r_1} + j't, t_{r_2} + j + 1't]\).

Observe that BGA and OPT do not accept any requests in the first period \([0, t_{r_1})\) and in the last period \([t_{r_2} + l't, \infty)\), and that the length of each period \( j \) (\( 1 < j < \gamma' \)), i.e., \((l_{j+1} - l_j)t\), is either \( t \), \( 2t \) or \( 3t \). By the partition rule (Algorithm 3), we have the following properties: if the length of period \( j \) is \( t \), i.e., \( l_{j+1} - l_j = 1 \), period \( j \) is sufficient; if the length of period \( j \) is \( 2t \), i.e., \( l_{j+1} - l_j = 2 \), either interval \( i \) is l-large or l-full and interval \( i + 1 \) is insufficient, or interval \( i \) is insufficient and interval \( i + 1 \) is r-full; if the length of period \( j \) is \( 3t \), i.e., \( l_{j+1} - l_j = 3 \), interval \( i \) and interval \( i + 2 \) are sufficient and interval \( i + 1 \) is r-full.

Let \( R_{(j)}'^{\gamma} \) denote the set of requests accepted by OPT that start in period \( j \), for \( 1 \leq j \leq \gamma' \). Let \( R_{(i)}^{\gamma} \) denote the set of requests accepted by BGA that start in period \( j \), for \( 1 \leq j \leq \gamma' \). By Observation 18, if interval \( i \) is insufficient and interval \( i - 1 \) is l-small, then interval \( i + 1 \) is r-full. By Lemma 15 and Lemma 19, if interval \( i \) is insufficient and interval \( i + 1 \) is insufficient, then interval \( i - 1 \) is l-full and interval \( i + 2 \) is r-full. From this it follows that an invariant of Algorithm 3 is that at the start of each iteration of the while-loop, either interval \( i \) is sufficient, or interval \( i \) is insufficient and interval \( i + 1 \) is r-full. Hence, the partition rule (Algorithm 3) is complete, i.e., in each iteration of the while-loop one of the if-cases applies.
We bound the competitive ratio of BGA by analyzing each period independently. As $R' = \bigcup_{j} R'_{(j)}$ and $R^* = \bigcup_{j} R^*_{(j)}$, it is clear that for any $\alpha \geq 1$, $P_{R^*}/P_{R'} \leq \alpha$ follows if we can show that $P_{R^*_{(j)}}/P_{R'_{(j)}} \leq \alpha$ for all $j$, $1 \leq j \leq \gamma'$.

According to Lemma 19, when $l_{j+1} - l_j = 2$, $P_{R^*_{(j)}}/P_{R'_{(j)}} \leq \frac{3}{5}$. According to Lemma 20, when $l_{j+1} - l_j = 3$, $P_{R^*_{(j)}}/P_{R'_{(j)}} \leq \frac{5}{3}$. By the partition rule, if $l_{j+1} - l_j = 1$, then period $j$ is sufficient, i.e., $|R^*_{(j)}| \geq \frac{3}{5}|R'_{(j)}|$, and hence $P_{R^*_{(j)}}/P_{R'_{(j)}} \leq \frac{5}{3}$. Since $P_{R^*_{(j)}} = P_{R'_{(j)}} = 0$ when $j = 1$ and $j = \gamma'$ (recall that by Observation 3, all requests accepted by BGA and $OPT$ do not start earlier than $t^*_R$ and do not start later than $t^*_R\gamma'$), the theorem follows. 

\section{Conclusion}

We have studied an on-line problem with $k$ servers and two locations that is motivated by applications such as car sharing and taxi dispatching. In particular, we have analyzed the effects that different constraints on the booking time of requests have on the competitive ratio that can be achieved. For all variants of booking time constraints we have given matching lower and upper bounds on the competitive ratio. The upper bounds are all achieved by the same balanced greedy algorithm (BGA) with different choices for the number of specified servers ($2k$). Interestingly, $k = 3$ servers suffice to achieve competitive ratio 1.5 (in the case of $kS2L-F$ with $a \geq t$ and $kS2L-V$ with $b_t \geq t$ and $b_u - b_l < t$), and $k = 5$ servers suffice to achieve competitive ratio $\frac{5}{3}$ (in the case of $kS2L-V$ with $b_t \geq t$ and $b_u - b_l \geq t$), and a larger number of servers does not lead to better competitive ratios.

In future work, it would be interesting to determine how the number of servers, the number of locations, and the constraints on the booking time affect the competitive ratio for the general car-sharing problem with $k$ servers and $m$ locations.

\textbf{References}


Packing Returning Secretaries

Martin Hoefer
Goethe University Frankfurt/Main, Germany
mhoefer@cs.uni-frankfurt.de

Lisa Wilhelmi
Goethe University Frankfurt/Main, Germany
wilhelmi@cs.uni-frankfurt.de

Abstract

We study online secretary problems with returns in combinatorial packing domains with \( n \) candidates that arrive sequentially over time in random order. The goal is to accept a feasible packing of candidates of maximum total value. In the first variant, each candidate arrives exactly twice. All \( 2n \) arrivals occur in random order. We propose a simple 0.5-competitive algorithm that can be combined with arbitrary approximation algorithms for the packing domain, even when the total value of candidates is a subadditive function. For bipartite matching, we obtain an algorithm with competitive ratio at least \( 0.5721 - o(1) \) for growing \( n \), and an algorithm with ratio at least \( 0.5459 \) for all \( n \geq 1 \). We extend all algorithms and ratios to \( k \geq 2 \) arrivals per candidate.

In the second variant, there is a pool of undecided candidates. In each round, a random candidate from the pool arrives. Upon arrival a candidate can be either decided (accept/reject) or postponed (returned into the pool). We mainly focus on minimizing the expected number of postponements when computing an optimal solution. An expected number of \( \Theta(n \log n) \) is always sufficient. For matroids, we show that the expected number can be reduced to \( O(r \log(n/r)) \), where \( r \leq n/2 \) is the minimum of the ranks of matroid and dual matroid. For bipartite matching, we show a bound of \( O(r \log n) \), where \( r \) is the size of the optimum matching. For general packing, we show a lower bound of \( \Omega(n \log \log n) \), even when the size of the optimum is \( r = \Theta(\log n) \).

2012 ACM Subject Classification Theory of computation → Online algorithms

Keywords and phrases Secretary Problem, Coupon Collector Problem, Matroids

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.65


Funding Supported by a grant from the German-Israeli Foundation for Scientific Research and Development (GIF).

1 Introduction

The secretary problem is a classic approach to study optimal stopping problems: A sequence of \( n \) candidates are arriving in uniform random order. Each candidate reveals its value only upon arrival and must be decided (accept/reject) before seeing any further candidate(s). Every decision is final – once a candidate gets accepted, the game is over. Moreover, no rejected candidate can be accepted later on. The goal is to find the best candidate. An optimal solution is to discard the first (roughly) \( n/e \) candidates. From the subsequent ones we accept the first that is the best one among the ones seen so far. The probability to hire the best candidate approaches \( 1/e \approx 0.37 \) when \( n \) tends to infinity.
The secretary problem and its variants have been popular since the 1960s. Significant interest in computer science emerged about a decade ago due to new applications in e-commerce and online advertising markets [2, 14]. For example, the classic secretary problem can be used to model a seller that wants to give away a single item, buyers arrive sequentially over time, and the goal is to assign the item to the buyer with highest value. More generally, online budgeted matching problems arise when search queries arrive over time, and the goal is to show the most profitable ads on the search result pages. The goal here is to design algorithms with good competitive ratio.

More recently, progress has been made towards a general understanding of online packing problems with random-order arrival, including matching [3, 20, 16], integer packing programs [22, 17], or independent set problems [13]. Most prominently, the matroid secretary problem has attracted a large amount of interest [2, 6]. Here the elements of a matroid arrive in uniform random order, and the goal is to construct an independent set with as high a value as possible. A central open problem in the area is the matroid secretary conjecture – is there a constant-competitive algorithm for every matroid in the random order model? The conjecture has been proved for a variety of subclasses of matroids [6]. Currently, the best-known algorithms for the general problem are $1/O(\log \log \text{rank})$-competitive [21, 8].

A strong assumption in the secretary problem is that every decision about a candidate must be made immediately without seeing any of the future candidates. Instead, in many natural admission scenarios candidates appear more than once, or they arrive and stay in the system for some time, during which a decision can be made. An interesting variant that captures this idea is the returning secretary problem [25]. Here each candidate is assigned two random time points from a bounded time interval. The earlier becomes the arrival time, the later the departure time. Hence, we can assume that each candidate arrives exactly twice, and all $2n$ arrivals occur in random order. The decision about acceptance of a candidate can be made between the first and the second arrival. More generally, for $k \geq 2$ each candidate chooses $k$ random points, arrives at the earliest and leaves at the latest point. In this case, there are $kn$ arrivals in random order. Vardi [25] showed an optimal algorithm for the returning secretary problem with $k = 2$, for which the probability of accepting the best candidate is about 0.768. For matroid secretary with $k = 2$ arrivals, a competitive ratio of 0.5, and for matching secretary a ratio $0.5625 - o(1)$ (with asymptotics in $n$) were shown.

In this paper, we significantly broaden and extend the results on the returning secretary problem towards general packing domains. We provide a simple algorithm that can be combined with arbitrary $\alpha$-approximation algorithms and yields competitive ratios of $0.5 \cdot \alpha$ for all subadditive packing problems, including matroids, matching, knapsack, independent set, etc. Moreover, we improve the guarantees for matching secretary and provide bounds that hold in expectation for all $n$. We extend all our bounds to arbitrary $k \geq 2$. In addition, we study a complementary variant in which the decision maker is allowed to postpone the decision about a candidate. In this case, the goal is to minimize the number of postponements to guarantee an optimal or near-optimal solution in the end. These problems can be cast as a set of novel coupon collector problems, and we provide guarantees and trade-offs for matroid, matching and knapsack postponement.

**Results and Contribution**

In the secretary problem with $k$ arrivals in Section 3, each candidate arrives exactly $k$ times. We propose a simple approach for general subadditive packing problems with returns, which can be combined with arbitrary $\alpha$-approximation algorithms. It yields a competitive ratio of $0.5 \cdot \alpha$ for $k = 2$, and $\alpha \cdot (1 - 2^{-k})$ for $k \geq 2$. 


For additive bipartite matching, we obtain a new algorithm that provides an improved competitive ratio of $0.5721 - o(1)$ for $k = 2$ with asymptotics in $n$. Moreover, we derive an algorithm with ratio $0.5459$ for $k = 2$ for every $n$. Both algorithms rely on exact solution of partial matching problems. The algorithms can be combined with faster $\alpha$-approximations for partial matchings, by spending at most an additional factor $\alpha$ in the competitive ratio. For the previous algorithm in [25], the algorithm description and proof of the ratio in the full version is slightly ambiguous. Our algorithm clarifies and slightly improves upon this by including the twice-arrived and rejected candidates during a sample phase when computing partial matchings. Their removal yields free nodes in the offline partition for matching in later rounds.

In the postponing secretary problem in Section 4, there is a pool of $n$ undecided candidates. In each round, a random candidate from the pool arrives. Upon arrival a candidate can be either decided (accept/reject) or postponed and returned into the pool. We strive to minimize the expected number of postponements to compute an optimal or near-optimal solution. Postponing everyone until all candidates are observed at least once is the coupon collector problem. Hence, with an expected number of $O(n \log n)$ postponements we reduce the problem to offline optimization. For general subadditive packing and an $\alpha$-approximation algorithm, a simple trade-off shows an $(1 - \varepsilon) \cdot \alpha$-approximation using $O(n \ln 1/\varepsilon)$ postponements.

Based on a property we term exclusion-monotonicity, we show significantly improved results when the desired solution has small cardinality. A bound of $O(r \log n)$ for the expected number of postponements holds when obtaining optimal solutions of size at most $r$ in additive matroids and bipartite matching, and greedy 2-approximations for knapsack. For matroids, we can further improve the bound to $O(r' \ln n/r')$, where $r' = \min(r, n - r)$. This upper bound is at most $n$, and the worst-case is attained for uniform matroids. We fully characterize the expected number of postponements of every candidate in uniform matroids when the optimal solution is to be obtained. Finally, we conclude the paper with a lower bound that in general we might need $\Omega(n \log \log n)$ postponements even with an optimal solution of cardinality $O(\log n)$. Due to space constraints, all missing proofs are deferred to the full version of this paper.

Further Related Work

The literature on secretary online variants of packing problems and online stochastic optimization has grown significantly over the last decade. We restrict the review to the most directly related results. For a survey of classic variants of the secretary problem, see [10].

The bipartite secretary matching problem was first studied in the context of transversal matroids [3], where a decision about accepting an arriving vertex into the matching needs to be taken directly, but matching edges can be decided in the end. Later works required that the edges must also be decided upon arrival [20]. The best algorithm for both variants obtains a competitive ratio of $1/e$ [16]. Most work in computer science has been devoted to the matroid secretary problem. Currently, the best algorithms obtain a competitive ratio $1/O(\log \log \text{rank})$ [21, 8]. It is an open problem if a constant competitive ratio can be shown. For a survey of work on classes of matroids and further developments see [6].

---

1 For example, the pseudo-code on page 12 does not become the algorithm for a single secretary when there is a single node in the offline partition. One would always accept the best secretary that arrived once in the sample phase. A better one arriving in later rounds is always rejected inside the for-loop. Also, the proof of Claim 5.6 seems to require both sides of the bipartite graph must have size $n$. 

ISAAC 2018
While above results are all for maximizing additive objective functions, recent work has started to consider submodular ones. For cardinality and matching constraints, constant-competitive algorithms exist for submodular secretary variants [18]. For matroids, there is a general technique to extend algorithms for additive objectives to submodular ones, which preserves constant competitive ratios [9].

Beyond matroids and matching, there are constant-competitive algorithms for knapsack secretary [1]. Prominent graph classes in networking applications allow good secretary algorithms for independent set [13]. The techniques for bipartite matching have been extended to secretary variants of combinatorial auctions and integer packing programs [22, 17]. Moreover, there are $1/O(\log n)$-competitive algorithms even in a general packing domain [23].

Additional model variants that have found interest are, for example, local secretary [5] (several decision makers try to simultaneously hire candidates based on local feedback), temp secretary [11] (candidates are hired only for a bounded period of time), or ordinal secretary [15, 24] (information available to the decision maker is only the total order of the candidates but not their numerical values).

Secretary postponement can be seen as a combinatorial extension of the coupon collector problem, a classic problem in applied probability. The elementary problem and its analysis are standard and discussed in many textbooks. The problem has many applications, and there is a plethora of variants that have been studied (see, e.g., [4, 12, 19]). To the best of our knowledge, however, the results for combinatorial packing problems derived in this paper have not been obtained in the literature before.

2 Packing Problems

We consider a packing problem, in which there is a set $N$ of $n$ candidates, and a set $S \subseteq 2^N$ of feasible solutions. $S$ is downward-closed, i.e. $S \subseteq T$ implies $T \subseteq S$. For most parts, we assume that the objective function $w : 2^N \rightarrow \mathbb{R}_{\geq 0}$ is additive, i.e., there is a non-negative value $w : N \rightarrow \mathbb{R}_{\geq 0}$ for each candidate, and $w(S) = \sum_{e \in S} w(e)$ for all $S \subseteq N$. More generally, we will sometimes assume the objective function $w$ is monotone and subadditive. If a packing problem has an $\alpha$-approximation algorithm, then for any $N' \subseteq N$ the algorithm guarantees an approximation ratio $\alpha \leq 1$ for maximizing $w$ over $S \cap 2^{N'}$.

In a secretary variant, we know the number $n$ upfront, and the candidates arrive in random order. Suppose a set $N_i$ of candidates has arrived in rounds $1, \ldots, i$ and candidate $e \in N \setminus N_i$ arrives in round $i + 1$. Then $e$ reveals all new feasible solutions with previously arrived candidates $(S \cap 2^{N \cup \{e\}}) \setminus (S \cap 2^{N_i})$ and their corresponding weight. In the additive case, this simply reduces to revealing the solutions and the weight $w(e)$.

We consider several specific variants. In matroid secretary, the set of candidates and the set of feasible solutions form a matroid. Upon arrival, a candidate reveals the new feasible solutions and their weights. In the additive variant with known matroid, all candidates and feasible solutions are known upfront. Candidates only reveal their weight upon arrival.

In (bipartite) matching secretary, there is a bipartite undirected graph $(N \cup V, E)$. The nodes in the offline partition $V$ are present upfront. The candidates in the online partition arrive sequentially. The feasible solutions are the matchings in the arrived subgraph. Upon arrival, a candidate reveals its incident edges and weights of the new feasible solutions. In the additive version, the arriving candidate reveals a weight per edge, and the weight $w(M)$ of a matching $M$ is the sum of edge weights. Upon accepting a candidate, the algorithm also has to decide which matching edge to include into $M$ (since otherwise it is matroid secretary with transversal matroid).
For (additive) knapsack secretary, an arriving candidate\(e\) reveals its weight \(w(e)\) and a size \(b(e) \geq 0\). The size \(B\) of the knapsack is known upfront. The feasible solutions are all subsets of candidates such that their total size does not exceed \(B\).

## 3 Secretaries with \(k\) Arrivals

Suppose that each candidate arrives exactly \(k\) times, and all these \(kn\) arrivals are presented in uniformly random order. Consider any subadditive secretary packing problem and the following simple algorithm. In the beginning, flip \(kn\) fair coins. The number of heads is the length of an initial sample phase. During the sample phase reject all candidates. Consider the set \(T\) of candidates that has appeared at least once and at most \(k-1\) times in the sample phase. Apply the \(\alpha\)-approximation algorithm to the instance based on \(S \cap 2^T\) to choose a feasible solution. Accept each candidate in the solution by the time of its \(k\)-th arrival.

\begin{proposition}
For any subadditive packing problem with an \(\alpha\)-approximation algorithm, the secretary problem with \(k\) arrivals allows an algorithm with approximation ratio
\[
\beta = \alpha \cdot \left(1 - \frac{1}{2^{k-1}}\right).
\]
\end{proposition}

\textbf{Proof.} Due to random order of arrival, we can simulate generation of \(T\) by attaching each of the \(kn\) coins to one arrival of one candidate. The arrival is in the sample phase if and only if the coin turns up heads. Then, the probability is \(1/2^k\) for each of the following events: (1) a given candidate never appears in the sample phase, and (2) a given candidate appears \(k\) times in the sample phase. \(T\) is distributed as if we would include each candidate independently with probability \(1 - \left(\frac{1}{2}\right)^{k-1}\).

Once \(T\) is created, we apply the \(\alpha\)-approximation algorithm to the instance based on \(S \cap 2^T\) to choose a feasible solution. Note that every candidate in \(T\) will appear at least once after the sample phase and therefore is available for acceptance by our algorithm. Each element in \(T\) is sampled independently from \(N\). Hence, as a simple consequence of subadditivity (see, e.g., [7, Proposition 2]), the value of the best feasible solution \(S_T^* \subseteq T\) has value \(w(S_T^*) \geq w(T \cap S^*) \geq \left(1 - \left(\frac{1}{2}\right)^{k-1}\right) \cdot w(S^*)\). By applying the \(\alpha\)-approximation algorithm to \(T\), we obtain a feasible solution \(S\) of value \(w(S) \geq \alpha \cdot w(S_T^*) \geq \alpha \cdot \left(1 - \left(\frac{1}{2}\right)^{k-1}\right) \cdot w(S^*)\).}

For secretary matching, we improve upon this by using a slightly more elaborate approach. The algorithm again samples and rejects a number of candidates that is determined by \(kn\) independent coin flips with a suitable probability \(p < 1\) (determined below). Hence, the length of the sample phase is distributed according to \(\text{Binom}(kn, p)\). At the end of the sample phase it computes a matching \(M_s\) using an \(\alpha\)-approximation algorithm for all known candidates and offline vertices \(V\). It accepts into \(M\) the edges incident to candidates with at most \(k-1\) arrivals in the sample. Each of them can be accepted upon their last arrival after the sample phase. The algorithm drops the edges from \(M_s\) incident to candidates that arrived \(k\) times in the sample. Let \(V_s \subseteq V\) be the unmatched offline nodes.

In the second phase, the algorithm follows ideas from [16, 25]. Upon arrival of a new candidate \(e\), the algorithm computes an \(\alpha\)-approximate matching \(M_e\) among \(V_s\) and all candidates with first arrival after the sample phase. If \(M_e\) contains an edge \((e, v)\) incident to \(e\), this edge is added into \(M\) if \(v\) is still unmatched. Otherwise the edge is discarded.

Since the algorithm can be combined with arbitrary \(\alpha\)-approximation algorithms for matching, it also applies to, e.g., the \(k\)-arrival variant of ordinal secretary matching [15].
Theorem 2. For secretary matching with 2 arrivals and any $\alpha$-approximation algorithm for offline matching with $\alpha \leq 1$, there is an algorithm with approximation ratio of $0.5721 \cdot \alpha - o(1)$. For $k$ arrivals, the ratio becomes at least $\alpha \cdot \left(1 - \frac{1}{2^k} + \frac{1}{2^k} - \frac{1}{2^k(2^k-1)^2}\right) - o(1)$.

Proof. By similar arguments as above, for each arrival of a secretary we can assume to flip a coin independently with probability $p < 1$ that determines if the arrival happens in the sample phase. Hence, each candidate has probability $p^k$ to arrive exactly $k$ times in the sample phase and $(1 - p)^k$ to never arrive in the sample phase. Let $M$ be the matching computed by the algorithm, $M_1$ the matching obtained right after the sample phase and $M_2$ the matching composed in the second phase. It holds $\mathbb{E}[w(M)] = \mathbb{E}[w(M_1)] + \mathbb{E}[w(M_2)]$.

For $M_1$ we interpret the random coin flips as a two-step process. First, for each candidate $n$ we flip a coin independently with probability $(1 - (1 - p)^k)$ whether the candidate arrives at least once in the sample phase. Then, we flip another independent coin with probability $p^k / (1 - (1 - p)^k)$ whether the candidate arrives $k$ times in the sample phase. The first set of coin flips determines the matching $M_s$ that evolves when we apply the $\alpha$-approximation algorithm right after the sample phase. Since every candidate is included independently we have $\mathbb{E}[w(M_s)] \geq (1 - (1 - p)^k) \cdot \alpha \cdot w(M^*)$. Afterwords, the second set of coin flips determines the candidates that are dropped from $M_s$. They are determined independently, so $\mathbb{E}[w(M_1)] = \mathbb{E}[w(M_2)] = (1 - (1 - p)^k - p^k) \cdot \alpha \cdot w(M^*)$.

We denote by $X$ the random number of candidates that arrived at least once during the sample phase. In the acceptance phase of the algorithm, we consider all $n - X$ candidates that have not arrived during the sample phase. Standard arguments [16, 25, 18] show that each of these newly arriving candidates contributes in expectation a value of $(\alpha \cdot w(S^*)) / n$. For the $\ell$-th first arrival of a new candidate, the probability that the edge $(v, v)$ suggested by the algorithm survives is the probability that the offline node $v \in V$ was not matched earlier, which is lower bounded by

$$\frac{p^k}{1 - (1 - p)^k} \cdot \prod_{r=X}^{\ell-1} \frac{r-1}{r} = \frac{p^k}{1 - (1 - p)^k} \cdot \frac{X - 1}{\ell - 1}.$$ 

Hence, the expected value for $M_2$ is at least

$$\mathbb{E}[w(M_2) \mid X] \geq \alpha \cdot w(M^*) \cdot \sum_{\ell=X}^{n} \frac{p^k}{1 - (1 - p)^k} \cdot \frac{X - 1}{\ell - 1} \cdot \frac{1}{n} \geq \alpha \cdot w(M^*) \cdot \frac{p^k}{1 - (1 - p)^k} \cdot \frac{X - 1}{n} \cdot \ln \frac{n}{X}.$$ 

For constants $p$ and $k$, standard Hoeffding bounds imply that $X = n(1 - (1 - p)^k) \pm o(n)$ with probability at least $1 - 1/n^c$ for suitable constant $c$ (see, e.g., [25]). Hence,

$$\mathbb{E}[w(M)] / w(M^*) \geq \alpha \left(1 - (1 - p)^k - p^k + p^k \cdot \ln \left(\frac{1}{1 - (1 - p)^k}\right)\right) - o(1),$$

where the asymptotics are in $n$. Numerical optimization shows that for $k = 2$ and $p \approx 0.49085$, the ratio becomes at least $0.57212 \cdot \alpha - o(1)$. See Table 1 for more numerical results.

Intuitively, the algorithm benefits from the unseen candidates after the sample phase and has a tendency to reduce the sample size. On the other hand, the candidates that come $k$ times within the sample phase create the set of free nodes in $V$ available for matching to later candidates. Overall, this leads to a small reduction in the sample size. For larger $k$ this effect vanishes since the number of candidates that appear never or $k$ times during the sample
phase both become exponentially small. The optimal sampling parameter quickly approaches 
\( p \to 0.5 \). This maximizes the profit from candidates that are available for optimization 
immediately after the end of the sample phase. Thereby, the improvement over the simple 
procedure in Proposition 1 becomes smaller.

More formally, we use \( \ln(1 + x) \geq x - x^2 \) in (1) and obtain

\[
\mathbb{E}[w(M)]/w(M^*) \geq \alpha \left( 1 - (1-p)^k - p^k \right) + \frac{p^k \cdot (1-p)^k}{1 - (1-p)^k} - \frac{p^k(1-p)^{2k}}{(1 - (1-p)^k)^2} - o(1) .
\]

Note that \( \ln(1 + x) \leq x \), so we deteriorate the expression only by the last negative term. For 
growing \( k \), the optimal value of \( p \) approaches 0.5 very quickly, and we bound

\[
\mathbb{E}[w(M)]/w(M^*) \geq \alpha \left( 1 - \frac{1}{2^k} - \frac{1}{2^k} \right) + \frac{\frac{1}{2^k} \cdot \frac{1}{2^k}}{1 - \frac{1}{2^k}} - \frac{\frac{1}{2^k} \cdot \frac{1}{2^k}}{(1 - \frac{1}{2^k})^2} - o(1) 
= \alpha \left( 1 - \frac{1}{2^{k-1}} + \frac{1}{2^{2k}} - \frac{1}{2^{2k}} \cdot \frac{1}{(2^{2k} - 2^{k+1} + 1)} \right) - o(1) .
\]

In contrast to [25], our algorithm computes an optimal (or \( \alpha \)-approximate) matching 
after the sampling phase for the set of all candidates that arrived during that phase (instead 
of the ones that arrived only once). All candidates that arrived \( k \) times are dropped. This 
creates free nodes of \( V \) to be matched to subsequently arriving candidates. The ratios depend 
asymptotically on \( n \), since the guarantee in the second phase relies on concentration bounds 
for \( X \), the number of candidates that arrive at least once in the sampling phase.

Alternatively, one can replace the second phase by recursively applying the sampling 
phase. More formally, after the sampling phase is done and matching \( M_t \) is added to \( M \), we 
apply the same sampling phase to \( V \) and the candidates that have not arrived so far. In this 
way, we can iterate the sampling step and re-apply it to the unseen candidates and left-over 
.nodes of the offline partition. The resulting ratios do not require concentration bounds.

\begin{corollary}
For secretary matching with 2 arrivals and any \( \alpha \)-approximation algorithm 
for offline matching with \( \alpha \leq 1 \), there is an algorithm with approximation ratio of \( 0.5459 \cdot \alpha \) 
for every \( n \geq 1 \). For \( k \) arrivals, the ratio becomes at least \( 1 - \frac{1}{2^{k+1}} + \frac{1}{2^{2k}} - \frac{2^{k-1}}{2^{2k} - 2^{k+1} - 1} \cdot \alpha \) 
for every \( n \geq 1 \).
\end{corollary}

4 Postponing Secretaries

Now suppose that for each arriving candidate the algorithm can decide (accept/reject) or 
postpone it. The goal is to compute an optimal or near-optimal solution with a small expected 
number of postponements. Consider any algorithm for the postponement problem. We 
cluster the execution into rounds. Round \( i \) are the arrivals from and including the \( i \)-th unique 
arrival (i.e., the \( i \)-th time a candidate arrives for the first time) and before the \( (i+1) \)-th 
unique arrival. Clearly, there are always \( n - 1 \) rounds in the execution of any algorithm.
If we simply postpone every candidate until we have seen all $n$ candidates, we have full information to make accept/reject decisions for all candidates. Then the problem reduces to the classic coupon collector problem, and the expected number of returns is $\Theta(n \log n)$. Our goal is to examine how we can improve upon this baseline.

We first consider a general result for subadditive packing. To reduce the expected number of returns to $\Theta(n)$, it is sufficient to sacrifice a constant factor in the approximation ratio.

▶ Proposition 4. For any $\varepsilon > 2/n$ and any subadditive packing problem with $\alpha$-approximation algorithm, there is an $\alpha \cdot (1 - \varepsilon)$-approximation algorithm with an expected number of postponements of $E[R] < n \cdot \ln(2/\varepsilon)$.

Proof. We postpone every candidate until round $\lceil n(1 - \varepsilon) \rceil$. Then, we run the $\alpha$-approximation algorithm on the subset of arrived candidates. By the same arguments as in Proposition 1, this yields an $\alpha(1 - \varepsilon)$-approximation.

Let $R^i$ be the number of postponements in round $i$. Clearly, by linearity of expectation, $E[R] = \sum_{i=1}^{n} E[R^i]$. In each round, the number of postponements is the number of rounds until we see the next unique arrival, and, hence, distributed according to a negative binomial distribution. Therefore, their expected number is

$$E[R] \leq \sum_{i=1}^{\lceil n(1-\varepsilon) \rceil} \left( \frac{n}{n-i} - 1 \right) = n \cdot \sum_{i=1}^{\lceil n(1-\varepsilon) \rceil} \frac{1}{n-i} - \lceil n(1-\varepsilon) \rceil$$

$$\leq n \cdot (\ln(n) - \ln(n\varepsilon - 1) - 1 + \varepsilon) \leq n \cdot (\ln(\varepsilon - 1/n)) < n \ln \left( \frac{2}{\varepsilon} \right).$$

4.1 Exclusion-Monotone Algorithms

We obtain significantly better guarantees for packing problems and algorithms with a monotonicity property. Consider a packing problem and any algorithm $\mathcal{A}$. We denote by $\mathcal{A}(T)$ the solution computed by $\mathcal{A}$ when applied to $T \subseteq N$.

▶ Definition 5. A sequence of subsets $(N_i)_{i \in \mathbb{N}}$ with $N_i \subseteq N$ is called inclusion-monotone if $N_i \subseteq N_j$ for all $i \leq j$. An algorithm $\mathcal{A}$ is called $r$-exclusion-monotone if for every inclusion monotone sequence there is a sequence of subsets $(D_i)_{i \in \mathbb{N}}$ with $\mathcal{A}(N_i) \subseteq D_i \subseteq N_i$, $|D_i| \leq r$ and $N_i \setminus D_i \subseteq N_j \setminus D_j$ for all $i \leq j$.

Intuitively, to determine its solution for any subset of available elements $N_i$, an $r$-exclusion-monotone algorithm $\mathcal{A}$ can restrict attention to a set $D_i$ of at most $r$ elements. Moreover, $\mathcal{A}$ is such that any element $e \in N_i \setminus D_i$ that is discarded must never be reconsidered when more elements become available.

This property is exhibited in a variety of important packing domains. For these problems we can obtain more fine-grained, significantly improved guarantees based on solution size.

---

2 For $\varepsilon \leq 2/n$, the bound remains $\Theta(n \log n)$ by simply observing all applicants and computing an $\alpha$-approximation.
Proposition 6. The following algorithms are r-exclusion-monotone.
- Optimal algorithm \(\text{OPT}\) for matroids. \(r\) is the rank of the matroid.
- Optimal algorithm \(\text{OPT}\) for bipartite matching. \(r\) is the maximum cardinality of any matching\(^3\).
- \text{GREEDY} 0.5-approximation algorithm for knapsack. Here \(r = |S| + 1\) with \(S\) a feasible packing of the knapsack with maximum cardinality.

Now consider candidates arriving in random order with postponements. Obviously, the set of arrived candidates forms an inclusion-monotone sequence. In our algorithm \(\text{MAINTAIN-A}\), we apply the r-exclusion-monotone algorithm \(A\) in the beginning of round \(i\) to the set \(N_i\) of arrived candidates. \(\text{MAINTAIN-A}\) immediately rejects any candidate as soon as it is not contained in \(D_i\). It keeps postponing the candidates in \(D_i\). Finally, \(\text{MAINTAIN-A}\) accepts the candidates in \(A(N)\) after the last round. Note that for the following result, \(\text{MAINTAIN-A}\) does not have to know \(n, r\) or any properties of the unseen candidates. The following guarantee significantly improves over the simple bound given in Proposition 4 when the solution is drawn from a small subset of elements.

Theorem 7. Consider a packing problem with an r-exclusion-monotone \(\alpha\)-approximation algorithm \(A\). The corresponding algorithm \(\text{MAINTAIN-A}\) computes an \(\alpha\)-approximation with an expected number of postponements \(\mathbb{E}[R] = \Theta(r \ln n/r')\), where \(r' = \min(r, n - r)\).

Proof. Consider the execution of the algorithm in rounds as discussed above. In each round, let \(U_i\) denote the number of candidates that are still undecided (i.e., either have not arrived or have been left undecided in earlier rounds). In round \(i\) we have seen exactly \(i\) candidates. Thus, given \(U_i\) undecided candidates, the expected number of postponements \(R_i\) in round \(i\) is given by a negative binomial distribution and amounts to

\[
\mathbb{E}[R_i | U_i] = \left(\frac{U_i}{n-i} - 1\right).
\]

To bound \(U_i\) we note that, trivially, \(U_i \leq n\). Moreover, the number of candidates that have arrived and are undecided is \(U_i - (n - i)\). Since \(\text{MAINTAIN-A}\) postpones only candidates in the set \(D_i\), we have that \(U_i - (n - i) \leq r\). This implies \(U_i \leq \min(n, n - i + r)\) and yields

\[
\mathbb{E}[R] \leq \sum_{i=1}^{r-1} \left(\frac{n}{n-i} - 1\right) + \sum_{i=r}^{n-1} \left(\frac{n-i+r}{n-i} - 1\right) = n \sum_{i=1}^{r-1} \frac{1}{n-i} - r + r \sum_{i=r}^{n-1} \frac{1}{n-i} \leq n \left(\frac{1}{n-r+1} + \ln\left(\frac{n-1}{n-r+1}\right)\right) + r \left(\frac{1}{r} - 1 + \ln\left(\frac{n-1}{r}\right)\right) = \left(2 + \frac{r-1}{n-r+1} - r\right) + n \ln\left(\frac{n-1}{n-r+1}\right) + r \ln\left(\frac{n-1}{r}\right).
\]

Clearly, the first term in the bracket is at most 1. For \(r \geq n-r+1\), the second term is larger than the third term and amounts to \(O(r \ln n/r')\). For \(r \leq n-r+1\), we upper bound

\[
n \ln\left(\frac{n-1}{n-r+1}\right) = n \ln\left(1 + \frac{r-2}{n-r+1}\right) \leq (r-2) + \frac{(r-2)(r-1)}{n-r+1} < 2r - 4.
\]

\(^3\) Recall that vertices in one partition arrive and get postponed, along with their incident edges. If single edges arrive and must be postponed individually, the property might not hold (c.f. Example 10 below).
Thus, the asymptotics are dominated by the third term, and \( E[R] = O(r \ln n/r') \). A similar calculation using elementary lower bounds shows that \( E[R] = \Omega(r \ln n/r') \). ▶

4.2 Matroids

We adjust MAINTAINOPT for known matroids, i.e. when the structure of the matroid is known upfront (only the weights of the elements are revealed). In this case, we can assume \( r \leq n/2 \), since for \( r \geq n/2 \) we can consider finding a minimum-weight basis in the dual matroid. We adjust algorithm MAINTAINOPT in the following way. Instead of postponing all elements in the current optimum until the end, we can accept some elements earlier. In particular, we can directly accept an element \( e \) as soon as there is no unseen element that can force \( e \) to leave the optimum solution. This allows to significantly improve the number of returns to below \( n \) for any rank of the matroid.

▶ Theorem 8. For the class of all matroids with rank \( r \), the expected number of postponements \( R \) in MAINTAINOPT with known matroid is maximized for the uniform matroid. It is bounded by \( E[R] = \Theta(r' \ln n/r') \), where \( r' = \min(r, n - r) \). For every matroid it holds that \( E[R] < n \).

Note that for any postponement problem, a simple calculation shows that the expected number of postponements of any single candidate can always be upper bounded by \( \ln(n+1) \). In contrast, the previous theorem shows that, on average, we need less than one postponement per candidate to compute even an optimal solution in matroids. However, they can be quite unbalanced over the candidates. We fully characterize the expected number of postponements in the uniform matroid with \( r \leq n/2 \). The worst candidate in the optimal solution (i.e., the \( r \)-th best candidate) asymptotically gets the largest expected number of postponements. The expected number is decreasing quickly for better and worse candidates.

▶ Theorem 9. For MAINTAINOPT with known uniform matroid of rank \( r \leq n/2 \), the expected number of postponements \( R_j \) of the \( j \)-th best candidate is bounded by

\[
E[R_j] \leq \begin{cases} 
\ln \left( \frac{n - j}{r - j + 1} \right) + O(1) , & \text{for } j = 1, \ldots, r, \\
\ln \left( \frac{j - 1}{j - r} \right) + O(1) , & \text{for } j = r + 1, \ldots, n.
\end{cases}
\]

Based on our experiments in Figure 1 the \( O(1) \) terms are small and even seem to vanish for large \( n \). The logarithmic function captures the number of postponements rather precisely.
For matroids, the number of postponements of MAINTAINOPT with known matroid is always at most \( n \). Instead, for bipartite matching the number of postponements of MAINTAINOPT must grow to \( \Theta(n \log n) \) when \( r \) becomes large, even if the graph is known.

**Example 10.** Consider a simple cycle of length \( 2n \) and number the vertices consecutively around the cycle. Suppose the \( r = n \) even vertices form the offline partition \( V \), and the \( n \) odd vertices arrive in random order. The edge weights can be arbitrary, but an adversary chooses them to be in \([1, 1 + \varepsilon]\). Then, unless we see all vertices, we cannot decide which of the two perfect matchings will be the optimal one. MAINTAINOPT needs to see all vertices to be able to decide the matching edges. We recover the coupon collector problem.

The example also applies when the edges of the bipartite graph are candidates that arrive in random order (rather than the vertices). In order to guarantee that an optimal solution is returned with probability 1 in the end, all \( 2n \) candidate edges need to remain undecided until the last unique arrival. This shows, in particular, that the bound of \( O(r' \ln \frac{n}{r'}) \) for MAINTAINOPT for known matroids cannot be extended to known intersections of matroids.

### 4.3 Exclusion-Monotonicity and Solution Size

For \( r \)-exclusion-monotone algorithms \( A \) the algorithm MAINTAIN-\( A \) needs at most \( O(r \ln n) \) postponements. One might hope that for any \( r \)-exclusion-monotone algorithm the parameter \( r \) is tied closely to the solution size of the algorithm. Then a large number of returns in MAINTAIN-\( A \) would be caused by \( A \) returning a solution with many elements. This, however, is not the case – even if we are guaranteed that the size of the optimal solution is \( \Theta(\log n) \), an expected number of \( \Omega(n \log \log n) \) postponements for MAINTAINOPT can be required.

**Theorem 11.** There is a class of instances of the independent set problem with every optimal solution of size \( |I^*| = 3 \ln n \), for which the expected number of postponements \( R \) in MAINTAINOPT is \( E[R] = \Omega(n \ln \ln n) \).

---

**References**

Packing Returning Secretaries

Simple $2^f$-Color Choice Dictionaries

Frank Kammer  
THM, University of Applied Sciences Mittelhessen, Germany  
frank.kammer@mni.thm.de

Andrej Sajenko  
THM, University of Applied Sciences Mittelhessen, Germany  
andrej.sajenko@mni.thm.de

Abstract  
A $c$-color choice dictionary of size $n \in \mathbb{N}$ is a fundamental data structure in the development of space-efficient algorithms that stores the colors of $n$ elements and that supports operations to get and change the color of an element as well as an operation choice that returns an arbitrary element of that color. For an integer $f > 0$ and a constant $c = 2^f$, we present a word-RAM algorithm for a $c$-color choice dictionary of size $n$ that supports all operations above in constant time and uses only $nf + 1$ bits, which is optimal if all operations have to run in $o(n/w)$ time where $w$ is the word size.

In addition, we extend our choice dictionary by an operation union without using more space.

2012 ACM Subject Classification  
Theory of computation → Data structures design and analysis

Keywords and phrases  
space efficient, succinct, word RAM

Digital Object Identifier  
10.4230/LIPIcs.ISAAC.2018.66

Acknowledgements  
After acceptance of the paper, we find out that Torben Hagerup has independently developed a colored choice dictionaries [10]. His choice dictionary supports any number of colors, but is more complicated even if the number of colors is a power of two. We gratefully thank Torben Hagerup for several helpful comments while we prepared our final version.

1 Introduction

Data is already an important factor for many companies and is likely to become even more decisive in the future. The most successful business enterprises today are often those that have figured out, better than their competitors, how to collect and use data. Therefore, it is no surprise that the number of companies that store petabytes of data in warehouses grows quickly. E.g., Walmart stored over 40 petabytes of data already in 2017.

In order to provide better support for structured access, data is often stored in highly redundant forms. In a data warehouse, a data item is usually considered as a point in a multidimensional space. A typical operation is to intersect a data cloud with a hyperplane obtained by fixing a single attribute to a specific value – the so-called slicing operation. If space is of no concern, it is easy to support slicing by storing for each possible such hyperplane the data points that it contains. But space matters, and we need dictionaries – data structures for storing and retrieving information – with low redundancy. More generally, we need algorithms that are fast but also treat memory as a scarce resource. Therefore, such algorithms (see, e.g., [1, 2, 4, 5, 6, 7, 8, 11, 12, 13]) become increasingly relevant. An implementation of several space-efficient algorithms can be found on GitHub [14].

1 Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 379157101.
In this paper, we focus on the improvement of a \((c\text{-color})\) choice dictionary, which is a fundamental data structure that is used in many of the algorithms listed above. A \textit{choice dictionary} [3, 11] manages the membership for \(n\) elements. The main characteristic of the dictionary is that it can retrieve an arbitrary member if one exists. If the dictionary can manage more than two states (member or not), it is called a \textit{\(c\)-color choice dictionary}.

\textbf{Definition 1.} A \(c\)-color choice dictionary is a data type that can be initialized with an arbitrary integer \(n \in \mathbb{N}\). Subsequently, it stores for each element \(e \in U = \{1, \ldots, n\}\) a color \(q \in Q = [0, c - 1]\), that is initially \(q = 0\) and supports the following standard operations:

- \texttt{setColor}_q(e) Sets the color of element \(e\) to \(q\).
- \texttt{color}(e) Returns the color \(q \in Q\) of element \(e\).
- \texttt{CHOICE}_q Returns an (arbitrary) element of \(U\) that has the color \(q\).

Another useful feature is an iterator that allows iterating over members of the dictionary and that can easily be used to support the slicing operation.

- \texttt{iterator.init}\(_q\) Returns an iterator for an iteration over the \(q\)-colored elements.
- \texttt{iterator.hasNext}\(_q\) Checks if the iterator can return a next element colored with color \(q\).
- \texttt{iterator.next}\(_q\) Returns the next element of the iterator over the \(q\)-colored elements.

If we talk about an \texttt{iterator}\(_q\) operation, then we mean the three operations \texttt{iterator.init}\(_q\), \texttt{iterator.hasNext}\(_q\), and \texttt{iterator.next}\(_q\). The iterator needs \(\Theta(\log n)\) bits. We call \(n\) the \textit{size} of the choice dictionary. We assume that \(n\) is given to the data structure with each call of an operation. In this paper, we extend our choice dictionary by another operation \texttt{union}\(_q,q'\) that recolors all \(q\) and \(q'\)-colored elements with one color. The outline of the paper is as follows. In the next section, we describe known techniques and sketch their usage in the paper. In Section 3, we describe an extension of Hagerup’s 2-color choice dictionary [9] to support \texttt{CHOICE} and \texttt{iterator} for both colors. Afterwards, we extend some word RAM tricks for parallel computations within one word described by Hagerup and Kammer [11] for our usage with several colors. In Section 5, we generalize our choice dictionary to \(2^f\) colors. We finally extend our choice dictionary by a \texttt{union} operation.
Figure 1 \( D \) shows the external view of the array and \( D \) represent the internal data structure. The variables \( i \) and \( j \) represent block numbers, \( \alpha, \beta, \gamma \) and \( \delta \) represent user defined words. The barrier \( b \) separates the array of blocks into a left and right area.

\[ \begin{array}{ccc|ccc|ccc|ccc} & \gamma & \delta & \cdots & 0 & 0 & \alpha & \beta & \cdots & 0 & 0 \\ \hline \text{left area} & \gamma & \delta & \cdots & j & \alpha & \cdots & i & \beta & \cdots & \text{right area} \end{array} \]

\[ \begin{array}{cccc} \end{array} \]

# Previous Techniques

In this section we summarize the ideas introduced by Katoh and Goto \([15]\) and Hagerup \([9]\) to implement our \( c \)-color choice dictionary.

## 2.1 Initializable Array

Katoh and Goto use a two level approach to support constant time initialization of an array consisting of \( n \) bits. They divided the array into \( O(n/w) \) blocks of \( O(w) \) bits each, number the blocks from left to right with \( 1, 2, 3, \ldots \), and distinguish between two block states. A block can be either initialized (with user defined values) or uninitialized (containing arbitrary values). The initialization of one block is done in \( O(1) \) time by setting its binary representation to zero. To implement the initialized array of \( n \) bits they determine if a block is initialized or uninitialized as follows – also sketched in Figure 1.

Partition the blocks using a barrier into a left area and a right area. All blocks in the left area are either initialized or have a so-called chain with an initialized block of the right area. A chain between an uninitialized block of the left area and an initialized block of the right area is created by writing the block number of each other in their first word. Because the initialized block contains values, but its first word is used to create the chain, they relocate the first word by storing it inside the second word of the chained uninitialized block. All blocks in the right area are uninitialized if they have no chain with a block of the left area.

To read a word of the array, determine to which block the word belongs to. Then determine if the block is inside the left or right area. If it is in the left area and has no chain, it is initialized and can be read and returned without further computations. If it has a chain, the block is uninitialized and zero can be returned. A block in the right area is uninitialized if it has no chain, so in this case a zero can be returned directly. If it has a chain, then the second word of the block can be read and returned directly, but for reading the first word, the chain must be followed to the block inside the left area and its second word must be returned.

Initially, the barrier is set before the first block and thus, the left area is empty and the array consists of only uninitialized blocks. If a block \( B \) of the left area is written, we must take care that no unintended chains are built, i.e., if the value in the first word points to an unchained block in the right area, then initialize that block with zeros. If a block \( B \) of the right area is written, it must be chained with an uninitialized block of the left area. Since we use the chain and unchain operations in the next sections, we describe them in detail.

The chain operation takes a block \( B \) of the left area and a block \( B' \) of the right area, relocates the first word of \( B' \) into the last word of \( B \) and writes the block number of \( B' \) in the first word of \( B \) and the block number of \( B \) in the first word of \( B' \). The unchain
operation takes a block \( B \) and returns an unchained block that needs to be chained. If \( B \) is not chained, initialize \( B \) with zeros and return it. If \( B \) is chained and in the left area, follow the chain pointer to a block \( B' \) of the right area, relocate the second word of \( B \) back to the first word of \( B' \) and initialize the block \( B \) with zeros. Finally, return \( B' \) to the caller. If \( B \) is in right area and \( B' \) chained to \( B \), call this operation again and return \texttt{UNCHAIN}(\( B' \)).

We say that a block \( B \) is \textit{connected to the left area} exactly if it is either inside the left area and unchained or inside the right area, but chained. Otherwise, it is called \textit{disconnected}.

The \texttt{CONNECT}(\( B \)) operation works as follows. We assume that \( B \) is not connected. Increase the barrier by one. This increases the left area and moves a block \( B^* \) from the right area into the left area. We differ two cases. In a first case \( B \) is in the right area before the increase of the barrier. If \( B = B^* \), then \( B \) is now connected, initialize all words of \( B \) with zeros and we are done. Otherwise, \( B^* \) is a candidate to create a chain with \( B \). However, it can be already chained. No matter if it is chained or not, call \( B' := \texttt{UNCHAIN}(B^*) \); note that the operation always returns an unchained block, possibly \( B^* \). Chain it with \( B \) by calling \texttt{CHAIN}(\( B', B \)). In a second case \( B \) is in the left area. Then by definition it must have a chain with a block \( B' \). To connect the block to the left area, the chain must be removed from \( B \), but \( B' \) still requires a chain. Thus, call \( B' = \texttt{UNCHAIN}(B) \), \( B'' := \texttt{UNCHAIN}(B^*) \) and \texttt{CHAIN}(\( B'', B' \)).

Writing blocks will cause blocks to connect to the left area and the left area to expand to the end of the array until every block belongs to it. Hagerup \cite{hagerup90} introduced a technique how to disconnect blocks, i.e, moving the barrier from right to left such that blocks can become uninitialized again. This technique was also shown in the second version of \cite{katoh97}.

The \texttt{DISCONNECT}(\( B \)) operation works exactly the other way around. We assume that \( B \) is connected. Decrease the barrier by one. This decreases the left area and moves a block \( B^* \) from the left area into the right area. If \( B \) is in the right area, call \( B' := \texttt{UNCHAIN}(B) \), \( B'' := \texttt{UNCHAIN}(B^*) \) and \texttt{CHAIN}(\( B', B'' \)). If \( B \) is in the left area, then it needs a chain to be disconnected, call \( B' = \texttt{UNCHAIN}(B^*) \) and \texttt{CHAIN}(\( B, B' \)).

To use the approach above we need to store the barrier. That requires \( O(w) \) extra bits of memory. To reduce the extra space needed to only 1 bit Katoh and Goto store the barrier inside the array as long not all blocks are initialized. If the array is fully initialized, i.e., all blocks are initialized, there is no need to chain blocks and the array is a normal array that can be read directly. One possible way to store the barrier in the array is to increase the block size to at least 3 words. Then the data of two chained blocks can be relocated such that the last word of every block inside the right area is always unused. (Either a block of the right area is uninitialized, i.e., all its word are unused, or is chained and therefore has one unused word left.) Since the left area expand to the right until the array is fully initialized, store the barrier inside the last unused word. Now an extra bit is used and set to 1 if the array is fully initialized and is a normal array and set to 0 if the array still has a right area of a barrier and therefore has to operate with blocks and chains.

### 2.2 Choice Dictionary with \texttt{CHOICE}_1 and \texttt{ITERATE}_1

Hagerup implemented \texttt{CHOICE}_1 and \texttt{ITERATE}_1 on an input consisting of \( n \) bits by using the techniques of Katoh and Goto. \texttt{CHOICE}_1 runs on \( O(w) \) bits in constant time using word RAM operations. Let us call a block \textit{non-empty} if it has at least one element (one bit) that is 1. Hagerup showed that all elements of uninitialized blocks of Katoh and Goto’s initialized array can be seen as zero bits and also that by uninitializing blocks that contain only zeros, the left area contains only non-empty blocks or have a chain with a non-empty block. To support \texttt{CHOICE}_1 one can pick an arbitrary block connected to the left area and to support \texttt{ITERATE}_1
one can move from the first block until the barrier and output the elements of the blocks connected to the left area. To output the elements of a block, again word RAM tricks can be used. Very recently, Hagerup [10] published a $c$-color choice dictionary for all $c \in \mathbb{N}$.

3 2-Color Choice Dictionary with $\text{CHOICE}_0$ and $\text{ITERATE}_0$

Hagerup supports $\text{CHOICE}_1$ and $\text{ITERATE}_1$ by managing the left area such that it only contains blocks that either have the color 1 or have a chain to a block with color 1. To support $\text{CHOICE}$ and $\text{ITERATE}$ on several colors the general idea is to manage such an area for each color, including color 0. In this section we assume to have only two colors $\{0, 1\}$, but since we want to support several colors later, we first give some definitions for an arbitrary number of colors before focusing on two colors.

We call a block that contains at least an element of color $q$ a $q$-block, a block that has no such element a $q$-free block, a block that contains only the color $q$ a $q$-full block and a block that contains all colors a full block. Moreover, we define two blocks as $q$-chained if their $q$th word contains the block number of each other and both blocks are separated by the barrier of color $q$. Keep in mind for the following algorithms that, if two blocks have a $q$-chain, the block in the left area of color $q$ must be a $q$-free block. Moreover, each $q$-block is connected to the left area of color $q$, and each $q$-free block is disconnected from this area.

To support several areas we change the data structure as follows: We store a barrier $b_q$ for each color $q \in \{0, \ldots, c - 1\}$ and initially let them point before the first block of the array, except for the barrier of color 0. The barrier of color 0 points after the last block of the array. The reason for this is to support constant-time initialization even if we have an uninitialized memory. We define that a block $B$ is uninitialized and is therefore defined as a block containing only the color 0 if $B$ fulfills the following condition. $B$ is in the right area for all colors $q \in \{1, \ldots, c - 1\}$ and in the left area for color 0 (i.e., $\max\{b_1, \ldots, b_{c-1}\} < B \leq b_0$) as well as $B$ has no $q$-chain for any color $q$. With this definition, initially all blocks are uninitialized and thus are 0-full blocks.

For the rest of the section, we focus on two colors. We assume now to have a block size of 3 words. Based on the ideas of Section 2.1, we next describe our invariant used to store the data in the blocks, which is also sketched in Figure 2. A block that has no chains simply stores its data unchanged – possibly, it is not initialized. A chained block uses its $q$th word ($q \in \{0, 1\}$) for a chain pointer of color $q$. Whenever a word of a chained block $B$ in the right area is used for any $q$-chain to a block $B'$ of the left area, then the user-data originally stored in the $q$th word of $B$ is stored in the last word of the block $B'$. Assume now that the $q$-chain points to a block of the right area. Then we know from the chain that $B$ is a $(1 - q)$-full block, and we do not need to store any user-data of the block.

It remains to show that no block has two chains to the right area since, otherwise, both chains want to store information in the last word of the block. Since we have only two colors, a block with a $q$-chain to the right must be a $(1 - q)$-full block. Since no block can be 0-full and 1-full simultaneous, no block can have two chains to the right.

Iterating over elements of a color $q$ can be done by iterating over the left area of color $q$ and by determining the block type of each block $B$. Either $B$ is a $q$-block or it is a $q$-free block chained with a $q$-block. Therefore, by using word RAM operations, we can output all elements of a color $q$ in linear time.

We now describe how to change colors of the dictionary such that the properties described above are maintained. If we change the color of an element of a block $B$ to a color $q$, we have to determine the block type first and then check if changing the color leads to a change of
the block type. Writing into a block changes its type only if we overwrite the last appearance of a color \( q' \) or introduce a new color \( q \).

Similar to Section 2 we use the operations \textsc{connect} and \textsc{disconnect} to change the block type, but redefine it that it works on a specific color \( q \): \textsc{chain} and \textsc{unchain} consider chains and barrier (thus, areas and block types) with respect to color \( q \). Whenever a color \( q \) is written to a \( q \)-free block \( B \), we connect the block \( B \) to the left area of \( q \). If the last appearance of a color \( q' \) has been overwritten after writing a color \( q \), we disconnect the block from the left area of color \( q' \).

Note that the chains and barriers of each color can be defined independently, and they do not interfere with the chains and barriers of other colors. Moreover, a block can change its type only if a color was introduced to it or has disappeared from it. After correcting the block type, write the color by modifying one word according to the new block type.

## 4 Word RAM Tricks

As long as we want to support only two colors, each element can be stored with only one bit. If the elements can have \( c = 2^f \) colors for some integer \( f > 1 \), we have to use \( f \) bits to store the color, which we combine into a field. Thus, a word can store \( w/f \) colors, each in one field, and \( f \) is the size of the field. For simplicity, we assume that \( w \) is a multiple of \( f \). If this is not the case, we may have to split the \( f \) bits of a color over two words, which does not change the asymptotic running time.

In the next section we want to modify several elements \( a_i \) in parallel that are located in fields part of one word in constant time. We next show some operations to realize the modifications.

The following lemma computes a bit mask consisting of 1 in the fields that have a value smaller than or equal to \( k \). Let \( m \) and \( f \) be given integers with \( 1 \leq m, f < 2^w \) and suppose that a sequence \( A = (a_1, \ldots, a_m) \) with \( a_i \in \{0, \ldots, 2^f - 1\} \) for all \( i \in \{1, \ldots, m\} \) is given in form of the \((mf)\)-bit binary representation of the integer \( x = \sum_{i=0}^{m-1} 2^f a_{i+1} \). Then the following holds:

\begin{lemma} [[11, Lemma 3.2c]] \end{lemma}

Given a parameter \( k \in \{0, \ldots, 2^f - 1\} \) in addition to \( x \) we can compute the integer \( z = \sum_{i=0}^{m-1} 2^{if} b_{i+1} \) in \( O(1 + mf/w) \) time, where \( b_i = 1 \) if \( k \geq a_i \) and \( b_i = 0 \) otherwise for \( i = 1, \ldots, m \).
We next use the lemma above to recolor all elements in a word of one color into another color quickly.

**Lemma 3.** Given two colors \( q \) and \( q' \) in addition to \( x \), we can compute the integer
\[
z = \sum_{i=0}^{m-1} 2^i b_i + 1,
\]
where \( b_i = a_i \) if \( a_i \neq q \) and \( b_i = q' \) otherwise for \( i = 1, \ldots, m \).

Proof. Apply the previous lemma twice on \( x \), first with parameter \( k = q \), then with \( k = q - 1 \), and subtract the second result from the first. We so get a vector \( v \) where the fields have a 1 exactly if the corresponding field in \( x \) has a \( q \). If \( q' < q \), then multiply \( v \) with \( q - q' \) and subtract the result from \( x \). Otherwise, \( q' > q \) and multiply \( v \) with \( q' - q \) and add the result to \( x \).

By applying the last lemma to all words in a block, we also get the following.

**Corollary 4.** Given the binary representation of a block consisting of the words \( x_1, \ldots, x_j \) \((j \in \mathbb{N})\) and two colors \( q \) and \( q' \), we can recolor all \( q \)-colored elements in the block with color \( q' \) in \( O(j) \) time.

In the next section, a block has to store pointers for chains even if only one color is missing in the block. This means that we need a word to store the pointer, but the elements of every word in the block can have several colors. As already described in [11], the idea is to “pack” the information in the words of the block, which is possible since a color is missing. We again start with an auxiliary lemma.

**Lemma 5 ([11, Part of the proof of Lemma 7.1]).** If all elements in \( x \) are different to a color \( q \), then we can pack \( c \) subsequent elements into \( cf - 1 \) bits (instead of \( cf \) bits). We so get a packed word, i.e., a word where every \((cf)\)th bit is not used to store the colors of the elements and therefore can be used to store other information. Both transformations (pack and unpack) run in \( O(c) \) time, but the unpack operation needs to know the color \( q \).

We now show how to store extra words within a block if one color is missing in the block. In the next section, the words are used to store pointers for chains.

**Lemma 6.** Let \( B \) be a block consisting of \( kc \log_2 c \) words and whose elements have a color in \( \{1, \ldots, c\} \). If all elements in the block have a color different to \( q \in \{1, \ldots, c\} \), then one can pack \( B \) in \( kc^2 \log_2 c \) time such that we can additionally store \( kw \) bits within the block. We combine the \( kw \) bits to \( k \) extra words. In a packed block, \( c \log_2 c \) time suffices to read and write an extra word, and given the color \( q \), to unpack a word of the block. Thus, we can unpack the whole block in \( k c^2 \log_2 c \) time.

Proof. To pack \( B \), run the algorithm from Lemma 5 on each word of the block. Note that the field size is \( f = \log_2 c \). We so get \( kw \) bits that are not used to store the colors of the elements. Similarly, if \( q \) is given, we can unpack the words.

If \( B \) is packed, every \( cf \) bit is free. To read and write a word \( x \) that is stored within the free bits, we first need to compute a vector \( v \) of \( w/(cf) \) fields of size \( cf \) with a 1 in all fields. We get \( v \) by using Lemma 2 with parameter \( k = 2^{cf} - 1 \) since all elements are \( \leq k \).

To read an extra word, first run bitwise-and operation with \( v \) to zero the bits between the free bits. Then combine the free bits part of several words with bitwise-or operations and bit-shifts. We so combine the free bits of several words into one word, which then can be returned as an extra word. See also Figure 3.

To write an extra word, we first apply \( v \) suitable shifted on the extra word to split the given word into \( cf \) words such that each field is either 0 or 1. After clearing the free bits in the block using again the bitwise-and operation with \( v \), we can use the bitwise-or to distribute the bits of the extra word to \( c \log_2 c \) words.


In Section 3, we focused on two colors. Now we increase the number of colors to $c = 2^f$, for an integer $f > 1$. We keep the concept of relocating words to create space for pointers. Again we exploit the fact that a block of the left area needs a chain if the block misses a color $q \in \{0, \ldots, c-1\}$. However, we need another idea to create space to store chain pointers because we cannot determine the original content of a block by knowing the missing color. (With more than two colors, a $q$-free block is not automatically $q'$-full.)

The idea is to pack a block that misses a color by using Lemma 6 such that a $q$-free block has extra space to store information. With a block size of $(2c + 1)c\log_2 c$ words, a packed block contains all its color information and has $2c + 1$ extra words to store extra information. For the time being, we assume in the following that a packed block contains the $2c + 1$ extra words at the beginning and then its packed color information. We use the first $c$ words to store up to $c$ chain pointers, the next $c$ words to store relocated words of a chained full block, and the last extra word to store the missed color that was used to pack the block. The rest of the block contains the packed color information.

We are not able to pack a full block since it has no redundancy. A full block located in the left area of all colors does not need to be chained. However, a full block $B$ in the right area of a color $q$ requires a chain with a block $B'$ that misses the color $q$. Thus, $B'$ is packed and has $2c + 1$ extra words. We use the $q$th word of $B'$ to store a chain pointer to $B$, relocate the $q$th word of $B$ into the $(c+q)$th word of $B'$ and store the chain pointer to $B'$ in the $q$th word of $B$. In contrast to the previous section, we only relocate words of a full block of the right area that requires a $q$-chain for some color $q$. If a block $B$ is a $q$-block in the right area of a color $q$ and misses some other color $q' \neq q$, $B$ is packed and can store all its colors and all its pointers.

To read a block we need to check if the block is packed (i.e., not full) and needs to be unpacked first. We know that a block $B$ is full exactly if, for each color $q$, either block $B$ is in the left area of the color $q$ and has no $q$-chain, or in the right area of color $q$ for that $B$ has a $q$-chain. To make this check we need to read chain pointers first. For each color $q \in \{1, \ldots, c-1\}$, we neither know if (1) a block is packed nor if (2) the $q$th word of a block stores colors or (3) a $q$-chain pointer. To find it out we have to check all $q$-chains. This is possible, since in all three cases (1) – (3), the $q$th chain is stored in the $q$th word. Thus, we only have to check if the $q$th word points to a block that points with its $q$th word back. Then $B$ is $q$-chained. After checking it for all colors, we know if we are in case (1), (2) or (3).

The rest of the read operation is simple. Either we can read the words in the block directly or we have to unpack a word of a block, i.e., we first read the $(2c + 1)$th extra word of the block where the color that was used to pack the block is located and use it to unpack the word such that the colors can be read.
Before we focus on the write operation, we start to discuss two problems and how we
deal with them. We first describe how to handle the uninitialized area between the barrier of
color 0 and the maximum barrier of all other colors, and second, how to avoid unintended
chains. The uninitialized area may contain arbitrary values and only chained blocks are
initialized in this area. These arbitrary values can be read as a pointer to a block that stores
color information and these color information can be interpreted as a back pointer. In this
situation the two blocks can have a chain that is unintended, and thus their block type
changes without intention. In general, an unintended chain may be created whenever a block
of the left area has been written with color information at a position where we also store
pointers for chains. The solution of the problem extends the ideas from Katoh and Goto.
We destroy unintended chains by checking if the word, where a chain pointer for a color \(q\)
may be stored, creates a chain with another block. If it does, we destroy such a chain by
writing \(n\) as an invalid chain pointer into the \(q\)th word of the unintended chained block in
the uninitialized area. We also store \(n\) inside every of the first \(c\) extra words of a packed
block that are not used to create chains. Since we use block numbers to create chains, the
value \(n\) can not point to a block and thus can never create a chain. From now on we ignore
the problems with uninitialized blocks and unintended chains.

We next describe the color change of an element \(e\) in a block \(B\) to color \(q\). Recall from
Section 3 that whenever we introduce a color \(q\) to a \(q\)-free block we connect the block to
the left area of color \(q\) and whenever a color \(q'\) disappears from a \(q'\)-block we disconnect the
block from the left area of color \(q'\).

To color \(e\) in \(B\) with \(q\), first determine if \(B\) is located in the left area of color \(q\), the block
type of \(B\), and if \(B\) is packed. Then, run the corresponding case.

- **\(B\) is in the left area:**
  - **\(B\) is a \(q\)-free block:** Unpack the block \(B\). Read the color \(q'\) of element \(e\), overwrite
    it with color \(q\) and connect \(B\) with the left area of color \(q\). Moreover, if \(B\) misses the
    color \(q'\), disconnect \(B\) from the left area of color \(q'\). We now differ two cases.
    (1) The block becomes a full block. Note that \(B\), as a full block, can not have chains
to the right area. For all \(q\)-chains to the left, with \(q' \in \{0, \ldots, c - 1\}\), relocate the \(q\)th
    word of the unpacked block \(B\) to the \((c + q)\)th word of the \(q\)-chained block.
    (2) \(B\) misses some color \(q^*\), possibly \(q^* = q'\). Pack the block using color \(q^*\).
  - **\(B\) is a \(q\)-block, but not full:** Unpack the block \(B\). Read the color \(q'\) of element \(e
    and overwrite it with color \(q\). If \(B\) contains no more \(q'\)-color elements, disconnect \(B
    from the left area of color \(q'\). Finally, use the color that was previously used to pack
    the block and pack it again.
  - **\(B\) is a full:** Thus, \(B\) is already unpacked. Read the color \(q'\) of element \(e\) and overwrite
    it with color \(q\). If \(B\) misses no color, then we are done. Otherwise, let \(q'\) be the missing
    color. Disconnect \(B\) from the left area of color \(q'\). We next want to pack \(B\). Before,
    we have to relocate back \(B\)'s first \(c\) words for all chains of \(B\) (all chains are to the left),
i.e. for all \(q\)-chains, with \(q' \in \{0, \ldots, c - 1\}\), move the \((c + q)\)th word in the \(q\)-chained
    block of \(B\) to \(q\)th word in \(B\). Then, pack \(B\) using color \(q'\).
- **\(B\) is in the right area:**
  - **\(B\) is a \(q\)-free block:** Unpack block \(B\). Read the color \(q'\) of element \(e\) and overwrite
    it with color \(q\). Connect \(B\) to the left area of color \(q\) and if \(B\) misses the color \(q'\),
    disconnect \(B\) from the left area of color \(q'\). We differ two cases.
    (1) \(B\) misses a color \(q^*\), possibly \(q^* = q'\). Pack the block using color \(q^*\).
    (2) Otherwise, \(B\) becomes a full block. Relocate \(B\)'s first \(c\) words into the chained
    blocks, i.e. for all \(q\)-chains, with \(q' \in \{0, \ldots, c - 1\}\) the \(q\)th word in \(B\) to the \((c + q)\)th
word of the block $\bar{q}$-chained with $B$.

- **$B$ is a $q$-block, but not full:** Unpack block $B$. Read the color $q'$ of element $e$ and overwrite it with color $q$. If color $q'$ is not present in $B$ anymore, disconnect it from the left area of color $q'$, and use the color that was previously used to pack the block.

- **$B$ is a full and unpacked:** Read the color $q'$ of element $e$ and overwrite it with color $q$. If color $q'$ is not present in $B$ anymore, disconnect it from the left area of color $q'$, and use the color that was previously used to pack the block.

Since we want to use Lemma 6, the extra words of a packed block are not a sequence of consecutive bits. They are distributed inside the block over every $(c \log_2 c)$th bit. Therefore, we can not store the pointers in a full block simply at the beginning and relocate one continuous word. Instead, we store our pointers in the same distributed way and the relocation saves the distributed bits.

To relocate the bits, we can simply use the read operation of Lemma 6 to get the distributed bits in compact words, which then can be relocated. Analogously, to read and write the pointer of a chain, we can use the read and write operation, respectively, of the lemma. Note that the read and write operation also works even if the words are not packed.

We now describe how we store the barriers in the dictionary as long as not every block contains all colors such that the dictionary requires only 1 extra bit. We increase the block size to $(3c + 1)c \log_2 c$ words, and increase so the number of extra words in a block by $c$, which allows us to store the $c$ barriers of all colors within one pair of chained blocks. Since all the barriers moves to the right, the rightmost block is always packed or has a chain to a packed block unless all blocks have all colors. Then, we do not require to store the barriers. The only information that we have to store is a one bit that is set to 1 exactly if every block contains all colors and the whole dictionary is a normal array that can be read by accessing the data directly.

The size $n$ of our $2^f$-color choice dictionary is necessary to decide if the barriers reached the end of the array or not. As long as we have chains, we can store the size $n$ in the first word and move the original content from there into some extra word. However, if all blocks are full – i.e., we have maximal entropy – we need $fn$ bits to store the colors. Thus, if we reintroduce the barriers again (a color disappears from a block), we need to know the size $n$ from the user. So, we assume that the user provides the size whenever an operation is called.

**Theorem 7.** For integers $f \geq 1$ and $c = 2^f$, there is a $c$-color choice dictionary that occupies $nf + 1$ bits of memory and supports all standard operations in $O(c^3 \log c)$ time.

### 6 Operation UNION

The **UNION** operation takes two colors $q$ and $q'$, iterates over all elements of one of these colors and recolors them with the other color. After recoloring the elements, it returns the color that was chosen. It is not user controlled which color is chosen as the new color.

To implement **UNION**($q$, $q'$) in amortized constant time we select the color that appears in fewer (or the same number of) blocks, say $q$. We know this by comparing the size of the left areas of $q$ and $q'$, i.e., by comparing the barriers of the two colors. If the extra bit of the choice dictionary is 1, the left areas are of equal size. Then, iterate over all blocks that are connected to the left area of $q$ and recolor all $q$ in $q'$ in that block in constant time per word (Corollary 4).
Theorem 8. For integers \( f \geq 1 \) and \( c = 2^f \), there is a \( c \)-color choice dictionary that occupies \( nf + 1 \) bits of memory and supports \( \text{setColor}_q \) for \( q \in \{0, \ldots, c-1\} \) in \( O(c^4 \log c) \) amortized time, all remaining standard operations in \( O(c^3 \log c) \) amortized time as well as a union operation in amortized constant time.

Proof. To define a potential function, let \( k_q \) be number of blocks that have a \( q \)-colored element, and let \( \sigma : \{0, \ldots, c-1\} \to \{0, \ldots, c-1\} \) be a permutation so that \( k_{\sigma(0)} \geq \cdots \geq k_{\sigma(c-1)} \). We now take \( \Phi = C \sum_{q=0}^{c-1} k_{\sigma(q)}q \) as the potential function for our amortized running time analysis where \( C > 0 \) is some integer defined below. In other words, each block with a \( q \)-colored element gives us a contribution of \( C\sigma^{-1}(q) \) to our potential function. Note that a necessary change of \( \sigma \) can be always done in such a way that \( \Phi \) does not change: Before \( k_{\sigma(q)} \geq k_{\sigma(q')} \) becomes wrong due to an increase of \( k_{\sigma(q')} \) or a decrease of \( k_{\sigma(q)} \) for some colors \( q, q' \in \{0, \ldots, c-1\} \) with \( \sigma(q) = \sigma(q') + 1 \), we have \( k_{\sigma(q)} = k_{\sigma(q')} \). Consequently, we can interchange the values of \( \sigma(q) \) and \( \sigma(q') \) without a change of \( \Phi \).

It is easy to see that the operations \( \text{COLOR} \) and \( \text{ITERATOR} \) do not change \( \Phi \). Let us now consider a call of operation \( \text{UNION} \) on colors \( q \) and \( q' \). Assume that \( k_q \geq k_{q'} \) and thus \( \sigma(q) < \sigma(q') \). \( \text{UNION} \) iterates through all blocks with color \( q' \) and recolors the \( q' \)-colored elements in the blocks. This can be done in \( O(k_{q'}c^4 \log c) = Ck_{q'} \) total time by choosing \( C = C'c^3 \log c \) for some constant \( C' > 0 \). Since \( \sigma(q) < \sigma(q') \), \( \Phi \) shrinks with each recoloring of all \( q' \)-colored elements in a block by \( C(\sigma(q') - \sigma(q)) \geq C \) since \( k_{q'} \) decreases by one and \( k_q \) increases by at most one. In total, \( \Phi \) shrinks by at least \( Ck_{q'} \) so that the amortized costs are 0.

Finally, note that the operation \( \text{setColor}_q \) increases \( \Phi \) by at most \( C(c-1) \) so that its amortized running time is \( O(Cc + c^3 \log c) = O(c^4 \log c) \).

We finally show that our amortized analysis of the last proof is tight, i.e., if we want to support an amortized constant-time \( \text{UNION} \) operation, the amortized time to color an element increases by a factor \( \Omega(c) \) to pay for the recoloring that are done by the \( \text{UNION} \) operations. We show this by constructing an instance where at least half of the elements are moved \( c-2 \) times, i.e., the \( \text{UNION} \) operation recolors the elements \( c-2 \) times. It suffices to consider only one fixed instance since we can easily scale the instance size by any factor \( z \); simply replace each word below by \( z \) copies having the same colors.

Let us assume that every word has one ball in each color that occurs in the word, and let us assign the balls to \( c-1 \) columns of a table as follows. A ball having color \( i \) is part of column \( i \) for each \( i \in \{1, \ldots, c-1\} \). Initially, we have only zero words. Those words have no balls at all. We next construct an instance by filling first column 1, then column 2, etc. and show by induction the following property: the fraction of balls in column \( i \) increases by \( \Phi \) at each step. Clearly, \( k_i = xk_{i-1} + 1 \). By choosing \( x \) large enough we get \( (x-1/x)(1 - (i-1)/2c) \) balls that are moved \( i-1 \) steps, i.e., the balls have visited columns 1, \ldots, \( i \). Clearly, \( k_i = xk_{i-1} + 1 \). By choosing \( x \) large enough we get \( (x-1/x)(1 - (i-1)/2c) \geq (1 - i/2c) ) \geq (1 - i/2c) \) and \( (x-1/x) \to 1 \) for \( x \to \infty \). This shows the property for column \( i \).
References


Succinct Data Structures for Chordal Graphs

J. Ian Munro
Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada
imunro@uwaterloo.ca

Kaiyu Wu
Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada
k29wu@uwaterloo.ca

Abstract

We study the problem of approximate shortest path queries in chordal graphs and give a \( n \log n + o(n \log n) \) bit data structure to answer the approximate distance query to within an additive constant of 1 in \( O(1) \) time.

We study the problem of succinctly storing a static chordal graph to answer adjacency, degree, neighbourhood and shortest path queries. Let \( G \) be a chordal graph with \( n \) vertices. We design a data structure using the information theoretic minimal \( \frac{n^2}{4} + o(n^2) \) bits of space to support the queries:

- whether two vertices \( u, v \) are adjacent in time \( f(n) \) for any \( f(n) \in \omega(1) \).
- the degree of a vertex in \( O(1) \) time.
- the vertices adjacent to \( u \) in \( (f(n))^2 \) time per neighbour
- the length of the shortest path from \( u \) to \( v \) in \( O(n f(n)) \) time

2012 ACM Subject Classification
Theory of computation → Shortest paths, Theory of computation → Data compression

Keywords and phrases Succinct Data Structure, Chordal Graph

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.67

Funding This work was supported by NSERC of Canada and the Canada Research Chairs Programme.

1 Introduction

Chordal graphs have a rich history of study. There were encountered in the study of Gaussian elimination of sparse matrices [15]. Chordal graphs have many equivalent characterizations including the absence of chordless cycles of length greater than 3, the existence of an perfect elimination order[16], the existence of a clique tree [4], and as the intersection graph of subtrees of a tree [18]. Tarjan et. al [16] gave a linear \( O(n + m) \) algorithm for recognizing chordal graphs with \( n \) vertices and \( m \) edges by computing a perfect elimination order. The structure of chordal graphs allows the computation of many otherwise NP-Hard problems to be solved in polynomial time. These include finding the largest clique or computing the chromatic number. Chordal graphs have found applications in many fields, including compiler construction [14] and databases [6].

We consider the problem of creating a data structure for a chordal graph through the lens of succinct data structures. The goal of succinct data structures is to store a set \( X \) of objects in the information theoretic minimal \( \log(|X|) + o(\log(|X|)) \) bits of space while still being able to efficiently support the relevant queries. Jacobson [10] is the first to consider...
space efficient data structures in this sense and he gave representations of bit vectors, trees and planar graphs. Further work in this area gave space minimal representations of dynamic trees [11], arbitrary graphs [8] and partial k-trees [7].

1.1 Related Work

Graphs are a fundamental combinatorial structure and it is no surprise that there are a lot of work in constructing space efficient data structure for different classes of graphs. Many classes of graphs have been considered, such as arbitrary graphs [8], partial k-trees [7], planar graphs [10] and separable graphs [2]. For chordal graphs, there has been work in the dynamic setting, focusing mainly on whether certain edge insertions/deletions preserve chordality [1, 9]. Banerjee et. al showed that insertions/deletions can be done in $O(\deg(u) + \deg(v))$ time where $(u, v)$ is the edge that is inserted/deleted. They also show a lower bound that $O(\log n)$ amortized time is required.

Singh et. al [17] gave an $O(n \log n)$ bit data structure for the problem of approximate distance queries in chordal graphs. Their result is a $2d + 8$ approximation, that is, the result of the query is anywhere between $d$ the actual distance and $2d + 8$.

1.2 Our Results

Our representation of a chordal graph is based on the clique tree [4]. We store a slight variation of the clique tree in the information theoretic minimal $n^2/4 + o(n^2)$ bits of space. We then augment this structure to support degree in $O(1)$, adjacency and neighbourhood in $O(f(n))$, $O(f(n)^2)$ respectively for any $f \in \omega(1))$ and distance queries in $O(n f(n))$. We then consider the problem of approximating the distance query and identify the necessary portions of the previous data structure required to answer this approximation to obtain a $n \log n + o(n \log n)$ bit data structure with $O(1)$ query time. The approximation is within 1 of the actual distance.

Finally we explore the close relationship between the distance query and the set intersection oracle problem, and show that heuristically, it is difficult to construct a data structure in the exact distance scenario.

2 Preliminaries

2.1 Graph Terminology

We will assume basic terminology from graph theory such as vertex, edge, tree, undirected graph, etc. We will denote an undirected graph as $G = (V, E)$ with vertex set $V$ and edge set $E$. We will denote an edge between vertices $u, v$ by $(u, v)$. The number of vertices as $n = |V|$ and the number of edges as $m = |E|$. As we will be dealing with multiple graph-like structures at the same time, we will use $V$ to denote the vertex set when the underlying graph is clear and $V(G)$ to denote the vertex set of graph $G$. To avoid confusion in discussing mapping a graph onto a tree, we will refer to vertices of trees as nodes. A clique of $G$ is a complete subgraph of $G$. Unless otherwise stated, our log are base 2.

2.2 Chordal Graph Structure

A graph $G$ is chordal if it does not contain any $C_k$, a cycle on $k$ vertices as an induced subgraph for any $k \geq 4$. We will assume that all our chordal graphs are connected, or if not we could treat each component separately. The well known result of Rose et al. [16]
states that this is equivalent to the existence of a perfect elimination order (PEO) of the vertices of $G$. A PEO of a chordal graph $G$ is an ordering $v_1, v_2, \ldots, v_n$ of $V$ such that the predecessor set $\text{pred}(v_i) = \{v_j; j < i, (v_i, v_j) \in E\}$ is a clique for every vertex $v_i \in V$. For simplicity, we will denote $v_i$ simply as $i$. Furthermore, one can construct a clique tree of $G$ using the maximal cliques of $G$. Here every node of the tree is assigned a maximal clique and the tree has the property that for every pair of cliques $K, K'$, $K \cap K'$ is contained in every clique along the path between the nodes corresponding to $K, K'$. This is equivalent to for every vertex $v \in V$, the set of cliques $v$ belongs to forms a contiguous subtree.

We will use a variant of the clique tree that has $n$ nodes constructed from the PEO, which we will denote as a tree decomposition of $G$. Let $T$ be a tree and $X : V(T) \rightarrow 2^V$ a function that assigns to each node of $T$ a subset of the vertices of $G$ such that:

- For every $v \in V$, the set of nodes $X^{-1}(v)$ is non-empty and is contiguous. We will call this the contiguous subtree property.
- For every pair of vertices $u, v \in V$, $(u, v)$ is an edge if and only if there is a tree node $T_w$ such that $u, v \in X(T_w)$.

Note clique trees satisfies these properties.

Define $B(i) = \text{pred}(i) \cup \{i\}$ which we will call the bag of $i$. Define the functions $s(i) = \min(\text{pred}(i))$ and $l(i) = \max(\text{pred}(i))$. It is easily seen that $\text{pred}(i) \subseteq B(l(i))$ since $l(i) \in \text{pred}(i)$ so it is adjacent to every element of $\text{pred}(i)$.

We will construct a tree decomposition $T$ from a PEO of $G$ inductively. The initial node is $T_1$ with $X(T_1) = \{1\} = \text{pred}(1) \cup \{1\} = B(1)$. Given a tree decomposition of $1, \ldots, i$, construct a tree decomposition of $1, \ldots, i + 1$ by creating a node $T_{i+1}$ with $X(T_{i+1}) = B(i)$ and connect $T_{i+1}$ to $T_{l(i+1)}$.

▶ Lemma 1. This construction is a tree decomposition of $G$.

Proof. The second condition is easily seen as for every edge $(i, j)$ with $i < j$ is in bag $X(T_j) = B(j)$. Conversely, every bag is a clique. For the first condition, each $T_i \in X^{-1}(i)$, so it is non-empty. Furthermore, since $\text{pred}(i) \subseteq B(l(i))$, it follows by induction that the set $X^{-1}(i)$ is contiguous for every $i$. ◀
Succinct Data Structures for Chordal Graphs

We will abuse notation and refer to both the tree node $T_i$ and the vertex $i$ as $i$ when the context is clear. We will naturally refer to $l(i)$ as the parent of $i$ and denote the tree decomposition constructed by $T_i$. We will build a second tree (not a tree decomposition) by setting the parent of $i$ as $s(i)$ and call this tree $T_s$.

2.3 Chordal Graph Enumeration

Wormald [19] showed that the number of connected labelled chordal graphs on $n$ vertices is asymptotic to $\sum_r \binom{n}{r}2^{r(n-r)} > \binom{n}{n/2}2^{n^2/4}$. To bound the number of unlabelled chordal graphs, we take into account the number of automorphisms and obtain a lower bound of $\binom{n}{n/2}2^{n^2/4}/n!$ unlabelled chordal graphs. Thus the information theoretic lower bound gives $\log(\binom{n}{n/2}2^{n^2/4}/n!) = n^2/4 - \Theta(n \log n)$ bits.

2.4 Succinct Structures Used

In this paper we will use both succinct trees and succinct bit vectors. While there have been work on further compressing bit vectors to zeroth order entropy [13], we only require the most basic form of bit vectors.

- **Lemma 2.** There is a succinct data structure for a bit vector $B$ of length $n$ using $n + o(n)$ bits of space that supports following operations in $O(1)$ time. [10]
  - $B[i]$: returns the bit at position $i$ of $B$.
  - $\text{rank}(i) = \sum_{k=1}^i B[k]$ the number of 1s at or before position $i$
  - $\text{select}(i) = j$ such that $B[j] = 1$ and $\text{rank}(j) = i$, is the position of the $i$-th 1.

- **Lemma 3.** There is a succinct data structure for a tree $T$ on $n$ nodes using $2n + o(n)$ bits of space that supports the following operations in $O(1)$ time. [11]
  - parent, $k$-th child
  - depth($i$), the depth of node $i$
  - level-ancestor($i$, $d$), the ancestor of node $i$ at depth $d$ in the tree
  - LCA($i$, $j$), the lowest common ancestor of nodes $i$, $j$

3 Representation, Adjacency and Neighbourhood

We will store the chordal graph as follows: for each vertex $i$, store a bit vector $W(i)$ of length $|B(l(i))|$ indicating which subset $\text{pred}(i)$ is of $B(l(i))$ equipped with rank and select operations. We also store 1 bit indicating whether this bit vector is the all 1s vector, and if so, store the length in $\log |B(l(i))|$ bits instead. We also store the trees $T_i, T_s$. We identify each vertex with the corresponding node in these trees. Unless otherwise stated, the tree relations such as parent, are in $T_i$.

- **Theorem 4.** This representation uses at most $n^2/4 + o(n^2)$ bits.

**Proof.** The main fact we will use is that $\text{pred}(i) \subseteq B(l(i))$ so that $|B(i)| \leq |B(l(i))| + 1$ and equality occurs only when $\text{pred}(i) = B(l(i))$. In this equality case, we need only $\log |B(l(i))| \leq \log n$ bits.

Consider the index $i$ such that bag size $|B(i)| = b$ is maximized. Since at each vertex, the bag size can only increase by 1 from its parent $l(i)$, there must be at least $b$ indices such that the above inequality is an equality, and we only need $\log n$ bits each in these indices, for a total of $b \log n \leq n \log n$ bits. In all other vertices $j$, we need to store at most $|B(l(j))| \leq |B(l(i))| = b$ bits (+$o(b)$ for the rank and select structures). Thus in total we need to store $(b + o(b))(n - b) + n \log n + O(n) \leq n^2/4 + o(n^2)$ bits. □
Figure 2: The label on each node is the bit vector $W(i)$. The ones that are all 1s are identified and only their lengths are stored, but for clarity, they are drawn out explicitly.

### 3.1 Adjacency Queries

This structure is enough to answer the following queries in $O(n)$ time:

- Given a vertex $i$ and an integer $k$, find the $k$-th smallest predecessor of $i$. We will call this $\text{decode}(i,k)$.

- Given two vertices $j < i$, determine whether $(i,j) \in E$ or equivalently, $j \in \text{pred}(i)$. We will call this $\text{adj}(i,j)$.

**Proof.** We will handle these queries recursively up the tree.

1. First find the index of the $k$-th predecessor in the parent $l(i)$ be $k' = \text{select}(W(i),k)$. The vertex we are looking for is thus the $k'$-th predecessor of $l(i)$.

2. If $l(i)$ is a predecessor of $i$ and it will be at index $|B(l(i))|$. This is the only predecessor that we know exactly, all others are relative. Hence if $k' = |B(l(i))|$ we report the answer being $l(i)$, otherwise we recursively call $\text{decode}(l(i),k')$. In the worst case, this will recurse $\text{depth}(i)$ times with $O(1)$ work per recursion, which could be as bad as $\Theta(n)$.

3. First note that every predecessor $j$ of $i$, their tree node must be an ancestor of the tree node corresponding to $i$ (in $T_l$). This is because predecessor set are taken as subsets of our ancestor’s predecessor sets. Thus it is necessary that $j$ is an ancestor of $i$ in $T_l$ and we can do this by $\text{LCA}(T_l,j,i) = j$, which if fails, we return false.

4. Next consider the path from $j$ to $i$, $j = p_0,p_1,\ldots,p_h = i$. We wish to calculate the index $k_{h-1}$ of $j$ in $B(l(i)) = B(p_{h-1})$ if it exists, at which point, we may determine whether it survived in the subset $\text{pred}(i)$ by checking the value of $W(i)[k_{h-1}]$. To do this, we know that the index of $j$ in $B(p_0)$ is simply $k_0 = |B(p_0)|$. Thus $j$ exists in $B(p_1)$ if $W(p_1)[k_0] = 1$, and its index in $B(p_1)$ is simply $k_1 = \text{rank}(W(p_1),k_0)$. We return false if $W(p_1)[k_0] = 0$. Thus we create the helper query $\text{adj}(i,j,k)$ which determines whether the $k$-th predecessor of $j$ is adjacent to $i$, with $\text{adj}(i,j) = \text{adj}(i,j,|B(j)|)$ and in the recursive case above, call $\text{adj}(i,p_1,k_1)$. We determine $p_1$ in $O(1)$ time by calling $\text{level-ancestor}(i,\text{depth}(j) + 1)$. This is $O(1)$ per recursive call and the number of calls is at most $\text{depth}(i)$ which could be as bad as $\Theta(n)$.  

\[<\]
To speed up the query times, we would need to store some additional information. For certain nodes, rather than storing its predecessors relative to its parents, we store them explicitly with a bit vector using \( n \) bits (that is we store the corresponding row of the adjacency matrix) along with a rank and select structure on this. To use \( o(n^2) \) bits, we can only store this information in \( o(n) \) of these nodes. Furthermore, we would like to select these nodes in an uniform manner, such that for the paths above, we will encounter these shortcut nodes with regularity. Formally, we would like to find a set of \( (o(n)) \) nodes such that every path of length \( k \) in \( T_i \) intersects one of these nodes. This is exactly the problem of \( k \)-path vertex cover. Bresar et al. [3] showed that while in general it is NP-hard, it is solvable on trees in linear time.

\[ \boxed{\text{Lemma 5. There is an algorithm that computes an optimal } k \text{-path vertex cover of a tree } T, \text{ of size at most } \frac{W(T)}{k} \text{ in linear time.}} \]

Let \( f = \omega(1) \) be any non-constant increasing function, for example, the inverse Ackermann function. Then by Lemma 5, we can find a set of at most \( \frac{n}{f(n)} = o(n) \) shortcut nodes such that every path in \( T_i \) contains one of these nodes. We may thus modify the above queries to cap the recursion depth.

- If \( i \) is a shortcut node, then the \( k \)-th predecessor of \( i \) is \( \text{select}(W(i), k) \). Thus the recursion depth is at most \( f(n) \). The time is thus \( O(f(n)) \).

- We follow the path to the root from \( i \) until we hit either \( j \) or a shortcut node. If it hit \( j \) first, then we continue as above, but with the recursion depth guaranteed to be less than \( f(n) \). If we hit a shortcut node \( p_0 \) first, check that \( j \) is a predecessor of \( p_0 \) by \( W(p_0)[j] = 1 \). If not, return false, otherwise call \( \text{adj}(i, p_0, \text{rank}(W(p_0), j)) \) since \( j \) is the \( \text{rank}(W(p_0), j) \)-th predecessor of \( p_0 \). Again the recursion depth is at most \( f(n) \) so the time is \( O(f(n)) \).

Thus we have the following result:

\[ \boxed{\text{Theorem 6. There is a data structure for chordal graphs on } n \text{ vertices that can answer adjacency queries in } f(n) \text{ time using } n^2/4 + n^2/f(n) + o(n^2) \text{ bits of space.}} \]

### 3.2 Degree, Neighbourhood queries

Degree queries are simple, since we may write the down the degree of every vertex in \( n \log n \) bits of space. For neighbourhood queries at vertex \( i \), we split it into two parts, those neighbours that are smaller than \( i \) and those that are greater.

- For the smaller neighbours, we simply query: at vertex \( i \), find the \( k \)-th predecessor of \( i \) for \( 1 \leq k \leq |\text{pred}(i)| \) - in other words, applying \( \text{decode}(i, k) \). This takes \( f(n) \) time per neighbour.

- For the larger neighbours at vertex \( i \), we store a bit vector of length \( (n - i)/f(n) \), with entry \( j \) being a 1 if there is a neighbour in range of vertices \( [i + (j - 1)f(n), i +jf(n)] \).

Total space is \( n^2/f(n) = o(n^2) \). We select each 1 from this bit vector and check adjacency for every vertex in the given range. Therefore, each neighbour will need at most \( f(n) \) adjacency queries, and thus we need \( f(n)^2 \) time per neighbour.

Thus we have the following:

\[ \boxed{\text{Theorem 7. There is a data structure for chordal graphs on } n \text{ vertices that can answer adjacency queries in } f(n) \text{ time and neighbourhood queries in } (f(n))^2 \text{ time per neighbour using } n^2/4 + O(n^2/f(n)) + o(n^2) \text{ bits of space.}} \]
4 Shortest Paths

Let $d(i,j)$ denote the distance between vertices $i,j$.

We would like to answer queries of the form:

- $sp(i,j) = i = p_0, p_1, \ldots, p_k = j$ a path from $i$ to $j$ of minimal length
- $dist(i,j) = k$ the length of the shortest path

We will show that these queries are difficult to answer, since they are a superset of adjacency queries. Thus we will look at approximate forms of these queries. We will use $asp, adist$ to denote the actual distance and $dist(i,j)$ to denote the result of our query. We would like $dist(i,j) = d(i,j)$ but in general this is difficult. Define the approximate forms of these queries as $asp, adist$.

4.1 Ancestor Case

We will first study the easy case, where $j < i$ is an ancestor of $i$.

Lemma 8. The following algorithm:

- repeatedly apply $s(.)$ to $i$ to obtain the sequence $i = p_0, p_1, \ldots, p_k$. This is equivalent to traverse the node to root path from $i$ in $T$.
- stop when $p_k > j$ but $p_{k+1} = s(p_k) \leq j$.
- if $adj(p_k, j)$ then $dist(i,j) = k+1$ and the path is $i = p_0, p_1, \ldots, p_k, p_{k+1} = j$. Otherwise, $dist(i,j) = k+2$ and the path is $i = p_0, p_1, \ldots, p_k, p_{k+1}, j$

correctly computes the distance (that is $dist(i,j) = d(i,j)$) and a shortest path between $i$ and $j$ given that $j$ is an ancestor of $i$.

Proof. We induct on the distance between $i$ and $j$.

If $d(i,j) = 1$, then $i$ and $j$ are adjacent. Furthermore, since $j \in pred(i)$, $s(i) \leq j$. Thus $i = p_0 = p_k$ and algorithm correctly gives $dist(i,j) = 1$.

Suppose that $d(i,j) = 2$, and let $i, h, j$ be a path from $i$ to $j$ with minimal $h$. Note that if $h > i$ then both $i, j \in pred(h)$ so they are adjacent, contradicts $d(i,j) = 2$. In all other cases, the algorithm will return the path $i, s(i), j$. We need to show that $s(i)$ and $j$ are adjacent. First note that $h > s(i)$ and they are adjacent by definition of $s(.)$. Thus the ordering must be either $i > h > s(i) > j$ or $i > j > h > s(i)$ or $i > j > h > s(i)$. In the first two orderings, $s(i), j \in pred(h)$. In the third ordering, since $s(i) \in pred(i)$, by the contiguous subtree property, it must exist in the bag along the entire path between $s(i), i$ which contains $j$. Thus $s(i) \in pred(j)$. In all these cases $s(i)$ is adjacent to $j$.

Now suppose that our algorithm is correct for distances $< k$. Let $d(i,j) = k$ and a shortest path be $i = p_0, p_1, \ldots, p_k = j$. We will show that there is a shortest path that begins with the step $i, s(i)$. Thus, $d(s(i), j) = k-1$ and a path for it can be found using the above algorithm. But the step $i, s(i)$ is the first step in the algorithm for distances $> 2$, so the combination of the two is exactly the output of the algorithm.

Essentially, we will replace $p_1$ by $s(i)$ and argue that the resulting sequence is still a path. Let $p_\alpha$ be the node such that $p_\alpha < i$. We claim that $\alpha = 1$ since otherwise, $p_{\alpha-1}$ is a descendant of $i$ and by the contiguous subtree property, $i$ is adjacent to $p_\alpha$. Thus we may replace the entire path $i, \ldots, p_\alpha$ by $i, p_\alpha$, contradicting minimality. Thus at each step of the shortest path, we must go to an ancestor. Let $p_2$ be the first node in the path such that $p_2 < s(i)$. Note that $p_{2-1} > s(i) > p_2$ is a path on the tree, and thus by the contiguous subtree property, $s(i)$ is adjacent to $p_2$ hence we may replace the path $i = p_0, p_1, \ldots, p_2$ with $i, s(i), p_2$.

\[\Box\]
To answer \( sp(i, j) \), we simply follow the algorithm, and traverse \( T_s \), so we can output the path in \( O(1) \) per vertex in the path. To answer \( dist(i, j) \) we would like to compute \( k \) efficiently. Denote \( i' = p_k \) in the algorithm. That is, \( p_k \) is the ancestor such that \( p_k > j \) but \( s(p_k) \leq j \). The only candidates are level-ancestor\(_T(i, depth(j))\) and level-ancestor\(_T(i, depth(j) + 1)\). Thus we may find \( i' \) in constant time. In both queries, we require 1 adjacency check in the final step. Finally, if we do not perform this check, we are able to answer the queries within 1. Thus we obtain:

\[ \text{Lemma 9. Using the data structure as before, and suppose that } j \text{ is an ancestor of } i \text{ in } T_l, \text{ then we can answer } sp(i, j) \text{ in } O(d(i, j) + f(n)) \text{ time and dist}(i, j) \text{ in } O(f(n)) \text{ time. We can answer asp}(i, j) \text{ in } O(d(i, j)) \text{ time and adist}(i, j) \text{ in } O(1) \text{ time such that } d(i, j) \leq |asp(i, j)| = adist(i, j) \leq d(i, j) + 1. \]

Furthermore, since we only need to traverse through \( T_l, T_s \) in the approximate queries, the space required is the two trees plus a table to identify the nodes that correspond to the same vertex. Thus the space required is \( n[\log n] + 4n + o(n) \) bits.

We note that \( \Theta(n \log n) \) bits is best possible for our idea of representing these two trees and the mapping between them. The mapping between them is equivalent to computing the function \( s(.) \). Since the order of the children in the trees does not matter, they are free trees. Consider the split graph with a size \( n/2 \) clique \( \{v_1, \ldots, v_{n/2}\} \), size \( n/4 \) independent set \( \{u_1, \ldots, u_{n/4}\} \) together with one child of each of the \( n/4 \) vertices in the independent set \( \{w_1, \ldots, w_{n/4}\} \). Furthermore, we have the freedom to allow \( s(u_i) \) to be any permutation of \( \{v_1, \ldots, v_{n/4}\} \) and also \( s(w_i) \) to be any permutation of \( \{v_{n/4+1}, \ldots, v_{n/2}\} \). Thus for any ordering of the children in the tree, we would have to store a permutation on \( n/4 \) elements. This requires \( \Theta(n \log n) \) bits.

### 4.2 General Case

Now we study the general case when \( i, j \) do not have the ancestor relation. We will reduce to the ancestor case by the following lemma:

\[ \text{Lemma 10. Consider the shortest node-path } P_G \text{ in } T_l, T_1, \ldots, T_s, \text{ For every shortest path } P_G \text{ from } i \text{ to } j \text{ in } G, \text{ and every node } T_w \text{ in } P_G, \text{ } B(T_w) \text{ contains a vertex of } P_G. \]

\[ \text{Proof. Note that if } (u, v) \in E \text{ then } X^{-1}(u) \cap X^{-1}(v) \neq \emptyset \text{ since there must be a bag that both } u, v \text{ belong to. Thus the set } \bigcup_{v \in P_G} X^{-1}(v) \text{ is a contiguous subtree of } T_l \text{ that contains both } T_i \text{ and } T_j. \text{ So in particular it contains the path } P_T. \]

Let \( h = LCA_T(i, j) \). Then \( B(h) \) contains a vertex \( x \) on a shortest path between \( i, j \). Thus, \( d(i, j) = d(i, x) + d(x, j) \). Furthermore, \( x \) is an ancestor of both \( i \) and \( j \).

\[ \text{Lemma 11. The algorithm dist}(i, j) \text{ (sp}(i, j)) \]
For each vertex $x \in B(h)$ compute $\text{dist}(x, i) + \text{dist}(x, j)$ (or $\text{sp}(x, i) \cup \text{sp}(x, j)$).

return the minimum sum (resp. path) among those calculated above.

Computes the distance (resp. a shortest path) between $i, j$. The time cost for $d(i, j)$ is $O(|B(h)| \cdot f(n)) = O(n \cdot f(n))$ and the time cost for $\text{sp}(i, j)$ is $O(|B(h)| \cdot (d(i, j) + f(n))) = O(n \cdot (d(i, j) + f(n)))$.

The time cost is dominated by the term $|B(h)|$ which is as bad as $O(n)$. It seems difficult to avoid performing the entire loop, so we will turn to approximation again.

Lemma 12. The algorithm $\text{adist}(i, j)$ ($\text{asp}(i, j)$):

- Compute $\text{dist}(h, i) + \text{dist}(h, j)$ (or $\text{sp}(h, i) \cup \text{sp}(h, j)$)

Gives an error of at most 2 in the distance between $i, j$. That is $d(i, j) \leq \text{dist}(i, j) \leq d(i, j) + 2$.

Proof. Let $x \in B(h)$ be the vertex that is in a shortest path between $i, j$. Consider the paths $h, \text{sp}(x, i)$ and $h, \text{sp}(x, j)$. These are paths between $h$ and $i, j$. Thus $d(h, i) \leq 1 + d(x, i)$ and $d(h, j) \leq 1 + d(x, j)$. Finally, we have $\text{adist}(i, j) = d(h, i) + d(h, j) \leq 2 + d(x, i) + d(x, j) = 2 + d(i, j)$.

4.3 Improved Bounds

We would like to improve the approximate distance algorithm in two ways: first, reduce the error to 1, and second, to use $\text{adist}$ as the subroutine rather than $\text{dist}$ as the subroutine. To do this, we would need to compare the computation steps that are done by both the approximate and the exact versions.

Let $x \in B(h)$ be the optimal vertex. We consider the computation of $\text{dist}(x, i)$ and $\text{dist}(h, i)$. In $\text{dist}(h, i)$, we compute $i'_h$ and depending on $(i'_h, h) \in E$ we return $\text{depth}(i'_h) + \text{depth}(i'_h) + 1$ or $+2$. In $\text{adist}(h, i)$ we always return $+2$ skipping the adjacency check. Now consider $\text{dist}(x, i)$. We compute $i'_x$ which is either $i'_h$ or an ancestor of $i'_h$. In the case that $i'_x = i'_h$, the worst case is that $(i'_x, x) \in E$, thus $\text{adist}(h, i) - \text{dist}(x, i) = 1$. If $i'_x$ is an ancestor of $i'_h$ then $\text{depth}(i'_x) < \text{depth}(i'_h)$ and $\text{adist}(h, i) - \text{dist}(x, i) \leq 0$ and we occur no error at all.

Therefore, we may replace $\text{dist}(h, i)$ by $\text{adist}(h, i)$ and obtain the same guarantees.

Next consider the case that both $(i'_h, h), (j'_h) \notin E$. This is exactly when the algorithm can potentially give an error of 2, since we may obtain an error of 1 in both branches. In this case, both $s(i'_h), s(j'_h) \in B(h)$ so they are adjacent. Therefore, instead of returning the path $i, i', s(i'), h, s(j'), j', \ldots, j$, we may return the path $i, i', s(i'), s(j'), j', \ldots, j$ and cut the error down to 1. Note that in $\text{adist}(i, h)$ we do not perform the adjacency check, so we will always contract the path. Thus we obtain the result:
Succinct Data Structures for Chordal Graphs

Theorem 13. The algorithm adist\((i,j)\):
\[
\text{return } \text{adist}(h, i) + \text{adist}(h, j) - 1
\]
approximates \(d(i, j)\) within 1 in \(O(1)\) time using \(n \log n + 4n + o(n)\) bits of space.

5 Relation to Set Intersection Oracle

We now consider the conditions in which our approximation algorithm is exact and when it incurs an error of 1. We argued above that we incur an error of 1 on both branches when there is \(x \in B(h)\) such that \((x, i'), (x, j') \in E\). Equivalently, \((x, i') \in E \iff x \in B(i')\). Thus \(x \in B(i') \cap B(j')\). Conversely, if no such \(x\) exists, we only incur an error of 1 on exactly one branch, and due to the adjustment our algorithm is exact.

Lemma 14. \(\text{adist}(i, j)\) incurs an error of 1 if and only if \(B(i') \cap B(j') \neq \emptyset\).

5.1 Set Intersection Oracle Problem

The set intersection oracle (SIO) problem is the following:

- \(\text{adist}(i,j):\) return \(\text{adist}(h, i) + \text{adist}(h, j) - 1\)
- \(\text{adist}(i,j)\) approximates \(d(i, j)\) within 1 in \(O(1)\) time using \(n \log n + 4n + o(n)\) bits of space.

Theorem 15. Let \(|U| = k\) and \(|U| = \omega(\log n)\) and \(|U| = O(n)\). Then \(O(n|U|)\) bits is necessary and sufficient to answer set intersection queries.

Proof. One direction is trivial. We may always represent a set with a length \(|U|\) bit vector, with position \(i = 1\) if \(i\) is in the set. To answer the queries, with compute the bit-wise-and of the bit vectors and check if it is the 0 vector. Therefore \(n|U|\) bits is sufficient to answer the query.

Conversely, consider the split graphs where we have a size \(n - k\) clique and a size \(k\) independent set. The neighbourhood of each of the vertices in the independent set is one of \(2^{n-k} - 1\) subsets of the clique (we omit the empty set and the entire set). Since there are \(k\) such vertices in the independent set, there are \(2^k(n-k)\) such graphs. Divide by \(n!\) to account for isomorphisms and we obtain a lower bound of \(k(n-k) - O(n \log n)\) bits required to represent these sets. Note that all of these split graphs have intersection number \(k + 1\). For \(k = o(n)\) and \(k \in \omega(\log n)\), \(k(n-k) - O(n \log n) = nk - o(nk)\). For \(k = cn\) for some constant \(c\), we require \((1 - c)nk = O(nk)\) bits.  

Lemma 14. \(\text{adist}(i, j)\) incurs an error of 1 if and only if \(B(i') \cap B(j') \neq \emptyset\).
The above does not try to optimize the query time of the data structure. To obtain an query time of $O(1)$, it is not known whether there is any non-naive data structure (storing the entire incidence matrix using $n^2/2$ bits) to solve the problem, even when $|U| = O(\log^4(n))$ (see conjecture 3 in [12]).

Next we show the close relationship between SIO and an exact distance oracle for chordal graphs. As shown above, it seems very difficult to construct an exact distance oracle that has query time $O(1)$ succinctly, using $n^2/4$ bits of space.

**Theorem 16.** Consider the SIO problem, with $n$ sets $S_i \subseteq U$ and $|U| = n$. Any solution using $B$ bits of space and has query time $t$ will yield an exact distance oracle for chordal graphs occupying $B + o(B)$ bits of space with query time $O(t)$. Conversely, any exact distance oracle for chordal graphs on $n$ nodes using $B(n)$ bits of space with query time $t(n)$ will yield a solution to the SIO problem on $n$ sets using $B(2n)$ bits of space and query time $t(2n)$.

**Proof.** WLOG assume $U = [n]$ and $S_i \neq \emptyset$ since if $S_i = \emptyset$ then $S_i \cap S_j = \emptyset$ for every $j$.

The lower bound implies that $B = \Omega(n^2)$. Suppose we have a SIO, then we simply store $B(i)$ for every vertex $i$. By lemma 14, we can detect when $\text{adist}(i, j)$ is wrong by applying the query $B(i') \cap B(j')$. Thus we have a chordal graph distance oracle using $B + o(B)$ bits of space with query time $t + O(1)$.

Conversely, consider the split graph on $2n$ vertices. Let the vertex set be $[n] \cup \{v_1, \ldots, v_n\}$ where $[n]$ is a clique and $\{v_1, \ldots, v_n\}$ is an independent set. It is easy to see that this graph is chordal. Let $N(v_i) = S_i$. Then $d(v_i, v_j) = 2$ or $3$ and $d(v_i, v_j) = 2 \Leftrightarrow S_i \cap S_j \neq \emptyset$. Thus we have reduced the SIO query to a exact distance query in chordal graphs on $2n$ vertices. ▶

References


Succinct Data Structures for Chordal Graphs


Tree Path Majority Data Structures

Travis Gagie
CeBiB – Center for Biotechnology and Bioengineering, Chile
School of Computer Science and Telecommunications, Diego Portales University, Chile
travis.gagie@gmail.com

Meng He
Faculty of Computer Science, Dalhousie University, Canada
mhe@cs.dal.ca

Gonzalo Navarro
CeBiB – Center for Biotechnology and Bioengineering, Chile
IMFD – Millenium Institute for Foundational Research on Data, Chile
Dept. of Computer Science, University of Chile, Chile
gnavarro@dcc.uchile.cl

Abstract
We present the first solution to $\tau$-majorities on tree paths. Given a tree of $n$ nodes, each with a label from $[1..\sigma]$, and a fixed threshold $0 < \tau < 1$, such a query gives two nodes $u$ and $v$ and asks for all the labels that appear more than $\tau \cdot |P_{uv}|$ times in the path $P_{uv}$ from $u$ to $v$, where $|P_{uv}|$ denotes the number of nodes in $P_{uv}$. Note that the answer to any query is of size up to $1/\tau$. On a $w$-bit RAM, we obtain a linear-space data structure with $O((1/\tau) \log^* n \log \log w \sigma)$ query time. For any $\kappa > 1$, we can also build a structure that uses $O(n \log^{[\kappa]} n)$ space, where $\log^{[\kappa]} n$ denotes the function that applies logarithm $\kappa$ times to $n$, and answers queries in time $O((1/\tau) \log \log w \sigma)$. The construction time of both structures is $O(n \log n)$. We also describe two succinct-space solutions with the same query time of the linear-space structure. One uses $2nH + 4n + o(n)(H + 1)$ bits, where $H \leq \log \sigma$ is the entropy of the label distribution, and can be built in $O(n \log n)$ time. The other uses $nH + O(n) + o(nH)$ bits and is built in $O(n \log n)$ time w.h.p.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Majorities on Trees, Succinct data structures

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.68


1 Introduction

Finding frequent elements in subsets of a multiset is fundamental in data analysis and data mining [12, 10]. When the sets have a certain structure, it is possible to preprocess the multiset to build data structures that efficiently find the frequent elements in any subset.

The best studied multiset structure is the sequence, where the subsets that can be queried are ranges (i.e., contiguous subsequences) of the sequence. Applications of this case include time sequences, linear-versioned structures, and one-dimensional models, for example. Data
structures for finding the mode (i.e., the most frequent element) in a range require time \( O(\sqrt{n}/\lg n) \), and it is unlikely that this can be done much better within reasonable extra space \([8]\). Instead, listing all the elements whose relative frequency in a range is over some fraction \( \tau \) (called the \( \tau \)-majorities of the range) is feasible within linear space and \( O(1/\tau) \) time, which is worst-case optimal \([1]\). Mode and \( \tau \)-majority queries on higher-dimensional arrays have also been studied \([13, 8]\).

In this paper we focus on finding frequent elements when the subsets that can be queried are the labels on paths from one given node to another in a labeled tree. For example, given a minimum spanning tree of a graph, we might be interested in frequent node types on the path between two nodes. Path mode or \( \tau \)-majority queries on multi-labeled trees could be useful when handling the tree of versions of a document or a piece of software, or a phylogenetic tree (which is essentially a tree of versions of a genome). If each node stores a list of the sections (i.e., chapters, modules, genes) on which its version differs from its parent’s, then we can efficiently query which sections are changed most frequently between two given versions. There has been little work previously on finding frequent elements on tree paths. Krizanc et al. \([15]\) considered path mode queries, obtaining \( O(\sqrt{n}\lg n) \) query time. This was recently improved by Durocher et al. \([11]\), who obtained \( O(\sqrt{n}/w\lg \lg n) \) time on a RAM machine of \( w = \Omega(\lg n) \) bits. Like on the more special case of sequences, these times are not likely to improve much. No previous work has considered the problem of finding path \( \tau \)-majority queries, which is more tractable than finding the path mode. This is our focus.

We present the first data structures to support path \( \tau \)-majority queries on trees of \( n \) nodes, with labels in \([1..\sigma]\), on a RAM machine. We first obtain a data structure using \( O(n \lg n) \) space and \( O((1/\tau)\lg \lg w \sigma) \) time (Theorem 3). Building on this result, we reduce the space to \( O(n) \) at the price of a very slight increase in the query time, \( O((1/\tau)\lg^* n \lg \lg w \sigma) \) (Theorem 6). We then show that the original query time can be obtained within very slightly superlinear space, \( O(n \log^{[\kappa]} n) \) for any desired \( \kappa > 1 \), where \( \log^{[\kappa]} n \) denotes the function that applies logarithm \( \kappa \) times to \( n \) (Theorem 7). Finally, we show that our linear-space data structure can be further compressed, to either \( 2nH + 4n + o(n)(H + 1) \) bits or \( nH + O(n) + o(nH) \) bits, where \( H \leq \log_{\sigma} \) is the entropy of the distribution of the labels in \( T \), while retaining the same query times of the linear-space data structure (Theorems 8 and 9). All our structures can be built in \( O(n \lg n) \) deterministic time; only the latter one requires that time only w.h.p. We close with a brief discussion of directions for future research. In particular, we describe how to adapt our results to multi-labeled trees.

Durocher et al. \([11]\) also considered queries that look for the least frequent elements and \( \tau \)-minorities on paths. In our extended version (https://arxiv.org/abs/1806.01804), we compress their data structure for \( \tau \)-minorities with only a very slight increase in query time.

## 2 Preliminaries

### 2.1 Definitions

We deal with rooted ordinal trees (or just trees) \( T \). Further, our trees are labeled, that is, each node \( u \) of \( T \) has an integer label \( \text{label}(u) \in [1..\sigma] \). We assume that, if our main tree has \( n \) nodes, then \( \sigma = O(n) \) (we can always remap the labels to a range of size at most \( n \) without altering the semantics of the queries of interest in this paper).

The path between nodes \( u \) and \( v \) in a tree \( T \) is the (only) sequence of nodes \( P_{uv} = \{u = z_1, z_2, \ldots, z_{k-1}, z_k = v\} \) such that there is an edge in \( T \) between each pair \( z_i \) and \( z_{i+1} \), for \( 1 \leq i < k \). The length of the path is \( |P_{uv}| = k \), for example the length of the path \( P_{uv} \) is 1. Any path from \( u \) to \( v \) goes from \( u \) to the lowest common ancestor of \( u \) and \( v \), and then from
there it goes to \( v \) (if \( u \) is an ancestor of \( v \) or vice versa, one of the two subpaths is empty).

Given a real number \( 0 < \tau < 1 \), a \( \tau \)-majority of the path \( P_{uv} \) is any label that appears (strictly) more than \( \tau \cdot |P_{uv}| \) times among the labels of the nodes in \( P_{uv} \). The path \( \tau \)-majority problem is, given \( u \) and \( v \), list all the \( \tau \)-majorities in the path \( P_{uv} \). Note that there can be up to \( \lceil 1/\tau \rceil \) such \( \tau \)-majorities.

Our results hold in the RAM model of computation, assuming a computer word of \( w = \Omega(\lg n) \) bits, supporting the standard operations.

Our logarithms are to the base 2 by default. By \( \lg^{[k]} n \) we mean the function that applies logarithm \( k \) times to \( n \), i.e., \( \lg^{[0]} n = n \) and \( \lg^{[k]} n = \lg(\lg^{[k-1]} n) \). By \( \lg^* n \) we denote the iterated logarithm, i.e., the minimum \( k \) such that \( \lg^k n \leq 1 \).

### 2.2 Sequence representations

A bitvector \( B[1..n] \) can be represented within \( n + o(n) \) bits so that the following operations take constant time: \( \text{access}(B, i) \) returns \( B[i] \), \( \text{rank}_b(B, i) \) returns the number of times bit \( b \) appears in \( B[1..i] \), and \( \text{select}_b(B, j) \) returns the position of the \( j \)-th occurrence of \( b \) in \( B \) \[9\].

If \( B \) has \( m \) 1s, then it can be represented within \( m \lg(n/m) + O(m) \) bits while retaining the same operation times \[18\]. Those structures can be built in linear time. Note the space is \( o(n) \) bits if \( m = o(n) \).

Analogous operations are defined on sequences \( S[1..n] \) over alphabets \([1..\sigma]\). For example, one can represent \( S \) within \( nH + o(n)(H + 1) \) bits, where \( H \leq \lg \sigma \) is the entropy of the distribution of symbols in \( S \), so that \( \text{rank} \) takes time \( O(\lg \lg \sigma) \), \( \text{access} \) takes time \( O(1) \), and \( \text{select} \) takes any time in \( \omega(1) \) \[4, Thm. 8\]. The construction takes linear time. While this \( \text{rank} \) time is optimal, we can answer partial rank queries in \( O(1) \) time, \( \text{prank}(S, i) = \text{rank}_{S[i]}(S, i) \), by adding \( O(n(1 + \lg H)) \) bits on top of a representation giving constant-time \( \text{access} \) \[3, Sec. 3\]. This construction requires linear randomized time.

### 2.3 Range \( \tau \)-majorities on sequences

A special version of the path \( \tau \)-majority queries on trees is range \( \tau \)-majority queries on sequences \( S[1..n] \), which are much better studied. Given \( i \) and \( j \), the problem is to return all the distinct symbols that appear more than \( \tau \cdot (j - i + 1) \) times in \( S[i..j] \). The most recent result on this problem \[2, 1\] is a linear-space data structure, built in \( O(n \lg n) \) time, that answers queries in the worst-case optimal time, \( O(1/\tau) \).

For our succinct representations, we also use a data structure \[1, Thm. 6\] that requires \( nH + o(n)(H + 1) \) bits, and can answer range \( \tau \)-majority queries in any time in \((1/\tau) \cdot \omega(1)\). The structure is built on the sequence representation mentioned above \[4, Thm. 8\], and thus it includes its support for \( \text{access} \), \( \text{rank} \), and \( \text{select} \) queries on the sequence. To obtain the given times for \( \tau \)-majorities, the structure includes the support for partial rank queries \[3, Sec. 3\], and therefore its construction time is randomized. In this paper, however, it will be sufficient to obtain \( O((1/\tau) \lg \lg \sigma) \) time, and therefore we can replace their \( \text{prank} \) queries by general \( \text{rank} \) operations. These take time \( O(\lg \lg \sigma) \) instead of \( O(1) \), but can be built in linear time\(^4\). Therefore, this slightly slower structure can also be built in \( O(n \lg n) \) deterministic time.

\(^4\) In fact, their structure \[1\] can be considerably simplified if one can spend the time of a general \( \text{rank} \) query per returned majority.
When a set has no structure, we can find its \( \tau \)-majorities in linear time. Misra and Gries [16] proposed an optimal solution that computes all \( \tau \)-majorities using \( O(n \log(1/\tau)) \) comparisons. When implemented on a word RAM over an integer alphabet of size \( \sigma \), the running time becomes \( O(n) \) [10].

### 2.4 Tree operations

For tree nodes \( u \) and \( v \), we define the operations \texttt{root} (the tree root), \texttt{parent}(\( u \)) (the parent of node \( u \)), \texttt{depth}(\( u \)) (the depth of node \( u \), 0 being the depth of the root), \texttt{preorder}(\( u \)) (the rank of \( u \) in a preorder traversal of \( T \)), \texttt{postorder}(\( u \)) (the rank of \( u \) in a postorder traversal of \( T \)), \texttt{subtreesize}(\( u \)) (the number of nodes descending from \( u \), including \( u \)), \texttt{anc}(\( u, d \)) (the ancestor of \( u \) at depth \( d \)), and \texttt{lca}(\( u, v \)) (the lowest common ancestor of \( u \) and \( v \)). All those operations can be supported in constant time and linear space on a static tree after a linear-time preprocessing, trivially with the exceptions of \texttt{anc} [6] and \texttt{lca} [7].

A less classical query is \texttt{labelanc}(\( u, \ell \)), which returns the nearest ancestor of \( u \) (possibly \( u \) itself) labeled \( \ell \) (note that the label of \( u \) needs not be \( \ell \)). If \( u \) has no ancestor labeled \( \ell \), \texttt{labelanc}(\( u, \ell \)) returns \texttt{null}. This operation can be solved in time \( O(\log \log w, \sigma) \) using linear space and preprocessing time [14, 21, 11].

### 2.5 Succinct tree representations

A tree \( T \) of \( n \) nodes can be represented as a sequence \( P[1..2n] \) of parentheses (i.e., a bit sequence). In particular, we consider the balanced parentheses representation, where we traverse \( T \) in depth-first order, writing an opening parenthesis when reaching a node and a closing one when leaving its subtree. A node is identified with the position \( P[i] \) of its opening parenthesis. By using \( 2n + o(n) \) bits, all the tree operations defined in Section 2.4 (except those on labels) can be supported in constant time [17].

This representation also supports \texttt{access}, \texttt{rank} and \texttt{select} on the bitvector of parentheses, and the operations \texttt{close}(\( P, i \)) (the position of the parenthesis closing the one that opens at \( P[i] \)), \texttt{open}(\( P, i \)) (the position of the parenthesis opening the one that closes at \( P[i] \)), and \texttt{enclose}(\( P, i \)) (the position of the rightmost opening parenthesis whose corresponding parenthesis pair encloses \( P[i] \); when \( P \) represents a tree, this parenthesis representation is the parent of the node that \( P[i] \) corresponds to).

Labeled trees can be represented within \( nH + 2n + o(n)(H + 1) \) bits by adding the sequence \( S[1..n] \) of the node labels in preorder, so that \texttt{label}(\( i \)) = \texttt{access}(\( S, \text{preorder}(i) \)).

### 3 An \( O(n \log n) \)-Space Solution

In this section we design a data structure answering path \( \tau \)-majority queries on a tree of \( n \) nodes using \( O(n \log n) \) space and \( O((1/\tau) \log \log w, \sigma) \) time. This is the basis to obtain our final results.

We start by marking \( O(\tau n) \) tree nodes, in a way that any node has a marked ancestor at distance \( O(1/\tau) \). A simple way to obtain these bounds is to mark every node whose height is \( \geq 1/\tau \) and whose depth is a multiple of \( 1/\tau \). Therefore, every marked node is the nearest marked ancestor of at least \( 1/\tau - 1 \) distinct non-marked nodes, which guarantees that there are \( \leq \tau n \) marked nodes. On the other hand, any node is at distance at most \( 2[1/\tau] - 1 \) from its nearest marked ancestor.
For each marked node \( x \), we will consider prefixes \( P_i(x) \) of the labels in the path from \( x \) to the root, of length \( 1 + 2^i \), that is,

\[
P_i(x) = \langle \text{label}(x), \text{label}(\text{parent}(x)), \text{label}(\text{parent}^2(x)), \ldots, \text{label}(\text{parent}^{2^i}(x)) \rangle
\]

(terminating the sequence at the root if we reach it). For each \( 0 \leq i \leq \lceil \lg \text{depth}(x) \rceil \), we store \( C_i(x) \), the set of \( (\tau/2) \)-majorities in \( P_i(x) \). Note that \( |C_i(x)| \leq 2/\tau \) for any \( x \) and \( i \).

By successive applications of the next lemma we have that, to find all the \( \tau \)-majorities in the path from \( u \) to \( v \), we can partition the path into several subpaths and then consider just the \( \tau \)-majorities in each subpath.

- **Lemma 1.** Let \( u \) and \( v \) be two tree nodes, and let \( z \) be an intermediate node in the path. Then, a \( \tau \)-majority in the path from \( u \) to \( v \) is a \( \tau \)-majority in the path from \( u \) to \( z \) (including \( z \)) or a \( \tau \)-majority in the path from \( z \) to \( v \) (excluding \( z \)), or in both.

**Proof.** Let \( d_{uz} \) be the distance from \( u \) to \( z \) (counting \( z \)) and \( d_{zv} \) be the distance from \( z \) to \( v \) (not counting \( z \)). Then the path from \( u \) to \( v \) is of length \( d = d_{uz} + d_{zv} \). If a label \( \ell \) occurs at most \( \tau \cdot d_{uz} \) times in the path from \( u \) to \( z \) and at most \( \tau \cdot d_{zv} \) times in the path from \( z \) to \( v \), then it occurs at most \( \tau \cdot (d_{uz} + d_{zv}) = \tau \cdot d \) times in the path from \( u \) to \( v \). ◀

Let us now show that the candidates we record for marked nodes are sufficient to find path \( \tau \)-majorities towards their ancestors.

- **Lemma 2.** Let \( x \) be a marked node. All the \( \tau \)-majorities in the path from \( x \) to a proper ancestor \( z \) are included in \( C_i(x) \) for some suitable \( i \).

**Proof.** Let \( d_{xz} = \text{depth}(x) - \text{depth}(z) \) be the distance from \( x \) to \( z \) (i.e., the length of the path from \( x \) to \( z \) minus 1). Let \( i = \lceil \lg d_{xz} \rceil \). The path \( P_i(x) \) contains all the nodes in an upward path of length \( 1 + 2^i \) starting at \( x \), where \( d_{xz} \leq 2^i < 2d_{xz} \). Therefore, \( P_i(x) \) contains node \( z \), but its length is \( |P_i(x)| < 1 + 2d_{xz} \). Therefore, any \( \tau \)-majority in the path from \( x \) to \( z \) appears more than \( \tau \cdot (1 + d_{xz}) > (\tau/2) \cdot (1 + 2d_{xz}) > (\tau/2) \cdot |P_i(x)| \) times, and thus it is a \( (\tau/2) \)-majority recorded in \( C_i(x) \). ◀

### 3.1 Queries

With the properties above, we can find a candidate set of size \( O(1/\tau) \) for the path \( \tau \)-majority between arbitrary tree nodes \( u \) and \( v \). Let \( z = \text{lca}(u, v) \). If \( v \neq z \), let us also define \( z' = \text{anc}(v, \text{depth}(z) + 1) \), that is, the child of \( z \) in the path to \( v \). The path is then split into at most four subpaths, each of which can be empty:

1. The nodes from \( u \) to its nearest marked ancestor, \( x \), not including \( x \). If \( x \) does not exist or is a proper ancestor of \( z \), then this subpath contains the nodes from \( u \) to \( z \). The length of this path is less than \( 2\lceil 1/\tau \rceil \) by the definition of marked nodes, and it is empty if \( u = x \).
2. The nodes from \( v \) to its nearest marked ancestor, \( y \), not including \( y \). If \( y \) does not exist or is an ancestor of \( z \), then this subpath contains the nodes from \( v \) to \( z' \). The length of this path is again less than \( 2\lceil 1/\tau \rceil \), and it is empty if \( v = y \) or \( v = z \).
3. The nodes from \( x \) to \( z \). This path exists only if \( x \) exists and descends from \( z \).
4. The nodes from \( y \) to \( z' \). This path exists only if \( y \) exists and descends from \( z' \).

By Lemma 1, any \( \tau \)-majority in the path from \( u \) to \( v \) must be a \( \tau \)-majority in some of these four paths. For the paths 1 and 2, we consider all their up to \( 2\lceil 1/\tau \rceil - 1 \) nodes as candidates. For the paths 3 and 4, we use Lemma 2 to find suitable values \( i \) and \( j \) so that \( C_i(x) \) and \( C_j(y) \), both of size at most \( 2/\tau \), contain all the possible \( \tau \)-majorities in those paths. In total, we obtain a set of at most \( 8/\tau + O(1) \) candidates that contain all the \( \tau \)-majorities in the path from \( u \) to \( v \).
To verify whether a candidate is indeed a $\tau$-majority, we follow the technique of Durocher et al. [11]. Every tree node $u$ will store $\text{count}(u)$, the number of times its label occurs in the path from $u$ to the root. We also make use of operation $\text{labelanc}(u, \ell)$. If $u$ has no ancestor labeled $\ell$, this operation returns $\text{null}$, and we define $\text{count}(\text{null}) = 0$. Therefore, the number of times label $\ell$ occurs in the path from $u$ to an ancestor $z$ of $u$ (including $z$) can be computed as $\text{count}(\text{labelanc}(u, \ell)) - \text{count}(\text{labelanc}(\text{parent}(z), \ell))$. Each of our candidates can then be checked by counting their occurrences in the path from $u$ to $v$ using

\[
(\text{count}(\text{labelanc}(u, \ell)) - \text{count}(\text{labelanc}(\text{parent}(z), \ell))) \\
+ (\text{count}(\text{labelanc}(v, \ell)) - \text{count}(\text{labelanc}(z, \ell))).
\]

The time to perform query $\text{labelanc}$ is $O(\lg \lg_w \sigma)$ using a linear-space data structure on the tree [14, 21, 11], and therefore we find all the path $\tau$-majorities in time $O((1/\tau) \lg \lg_w \sigma)$.

The space of our data structure is dominated by the $O(\lg n)$ candidate sets $C_i(x)$ we store for the marked nodes $x$. These amount to $O((1/\tau) \lg n)$ space per marked node, of which there are $O(\tau n)$. Thus, we spend $O(n \lg n)$ space in total.

**Theorem 3.** Let $T$ be a tree of $n$ nodes with labels in $[1..\sigma]$, and $0 < \tau < 1$. On a RAM machine of $w$-bit words, we can build an $O(n \lg n)$ space data structure that answers path $\tau$-majority queries in time $O((1/\tau) \lg \lg_w \sigma)$.

### 3.2 Construction

The construction of the data structure is easily carried out in linear time (including the fields $\text{count}$ and the data structure to support $\text{labelanc}$ [11]), except for the candidate sets $C_i(x)$ of the marked nodes $x$. We can compute the sets $C_i(x)$ for all $i$ in total time $O(\text{depth}(x))$ using the linear-time algorithm of Misra and Gries [16] because we compute $(\tau/2)$-majorities of doubling-length prefixes $P_i(x)$. This amounts to time $O(mt)$ on a tree of $t$ nodes and $m$ marked nodes. In our case, where $t = n$ and $m \leq \tau n$, this is $O(\tau n^2)$.

To reduce this time, we proceed as follows. First we build all the data structure components except the sets $C_i(x)$. We then decompose the tree into heavy paths [20] in linear time, and collect the labels along the heavy paths to form a set of sequences. On the sequences, we build in $O(t \lg t)$ time the range $\tau$-majority data structure [2, 1] that answers queries in time $O(1/\tau)$. The prefix $P_i(x)$ for any marked node $x$ then spans $O(\lg t)$ sequence ranges, corresponding to the heavy paths intersected by $P_i(x)$. We can then compute $C_i(x)$ by collecting and checking the $O(1/\tau)$ $(\tau/2)$-majorities from each of those $O(\lg t)$ ranges.

Let the path from $x$ to the root be formed by $O(\lg t)$ heavy path segments $\pi_1, \ldots, \pi_k$. We first compute the $O(1/\tau)$ $(\tau/2)$-majority in the sequences corresponding to each prefix $\pi_1, \ldots, \pi_k$: For each $\pi_j$, we (1) compute its $2/\tau$ majorities on the corresponding sequence in time $O(1/\tau)$, (2) add them to the set of $2/\tau$ majorities already computed for $\pi_1, \ldots, \pi_{j-1}$, and (3) check the exact frequencies of all the $4/\tau$ candidates in the path $\pi_1, \ldots, \pi_j$ in time $O((1/\tau) \lg \lg_w \sigma)$, using the structures already computed on the tree. All the $(\tau/2)$-majorities for $\pi_1, \ldots, \pi_j$ are then found.

Each path $P_i(x)$ is formed by some prefix $\pi_1, \ldots, \pi_j$ plus a prefix of $\pi_{j+1}$. We can then carry out a process similar to the one to compute the majorities of $\pi_1, \ldots, \pi_{j+1}$, but using only the proper prefix of $\pi_{j+1}$. The $O(\lg t)$ sets $C_i(x)$ are then computed in total time $O((1/\tau) \lg t \lg \lg_w \sigma)$. Added over the $m$ marked nodes, we obtain $O((1/\tau)m \lg t \lg \lg_w \sigma)$ construction time.

**Lemma 4.** On a tree of $t$ nodes, $m$ of which are marked, all the candidate sets $C_i(x)$ can be built in time $O((1/\tau)m \lg t \lg \lg_w \sigma)$. 

The construction time in our case, where \( t = n \) and \( m \leq \tau n \), is the following.

\[ \text{Corollary 5. The data structure of Theorem 3 can be built in time } O(n \log n \log \log \sigma). \]

\section{A Linear-Space (and a Near-Linear-Space) Solution}

We can reduce the space of our data structure by stratifying our tree. First, let us create a separate structure to handle unary paths, that is, formed by nodes with only one child. The labels of upward maximal unary paths are laid out in a sequence, and the sequences of the labels of all the unary paths in \( T \) are concatenated into a single sequence, \( S \), of length at most \( n \). On this sequence we build the linear-space data structure that solves range \( \tau \)-majority queries in time \( O(1/\tau) \) \cite{2,1}. Each node in a unary path of \( T \) points to its position in \( S \). Each node also stores a pointer to its nearest branching ancestor (i.e., ancestor with more than one child).

The stratification then proceeds as follows. We say that a tree node is \textit{large} if it has more than \( (1/\tau) \log n \) descendant nodes; other nodes are \textit{small}. Then the subset of the large nodes, which is closed by \texttt{parent}, induces a subtree \( T' \) of \( T \) with the same root and containing at most \( \tau n/\log n \) leaves, because for each leaf in \( T' \) there are at least \( (1/\tau) \log n - 1 \) distinct nodes of \( T \) not in \( T' \). Further, \( T - T' \) is a forest of trees \( \{F_i\} \), each of size at most \( (1/\tau) \log n \).

We will use for \( T' \) a structure similar to the one of Section 3, with some changes to ensure linear space. Note that \( T' \) may have \( \Theta(n) \) nodes, but since it has at most \( \tau n/\log n \) leaves, \( T' \) has only \( O(\tau n/\log n) \) branching nodes. We modify the marking scheme, so that we mark exactly the branching nodes in \( T' \). Spending \( O((1/\tau) \log n) \) space of the candidate sets \( C_i(x) \) over all branching nodes of \( T' \) adds up to \( O(n) \) space.

The procedure to solve path \( \tau \)-majority queries on \( T' \) is then as follows. We split the path from \( u \) to \( v \) into four subpaths, exactly as in Section 3. The subpaths of type 1 and 2 can now be of arbitrary length, but they are unary, thus we obtain their \( 1/\tau \) candidates in time \( O(1/\tau) \) from the corresponding range of \( S \). Finally, we check all the \( O(1/\tau) \) candidates in time \( O((1/\tau) \log \log \sigma) \) as in Section 3.

The nodes \( u \) and \( v \) may, however, belong to some small tree \( F_i \), which is of size \( O((1/\tau) \log n) \). We preprocess all those \( F_i \) in a way analogous to \( T \). From each \( F_i \) we define \( F_i' \) as the subtree of \( F_i \) induced by the \texttt{parent}-closed set of the nodes with more than \( (1/\tau) \log \log n \) descendants; thus \( F_i' \) has \( O(|F_i|/\tau \log \log n) \) branching nodes, which are marked. We store the candidate sets \( C_i(x) \) of their marked nodes \( x \), considering only the nodes in \( F_i' \).

If the candidates were stored as in Section 3, they would require \( O((1/\tau) \log \sigma) \) bits per marked node. Instead of storing the candidate labels \( \ell \) directly, however, we will store \texttt{depth}(\( y \)), where \( y \) is the nearest ancestor of \( x \) with label \( \ell \). We can then recover \( \ell = \text{label}(\text{anc}(x, \text{depth}(\( y \)))) \) in constant time. Since the depths in \( F_i \) are also \( O((1/\tau) \log n) \), we need only \( O(\log((1/\tau) \log n)) \) bits per candidate. Further, by sorting the candidates by their \texttt{depth}(\( y \)) value, we can encode only the differences between consecutive depths using \( \gamma \)-codes \cite{5}. Encoding \( k \) increasing numbers in \([1..m]\) with this method requires \( O(k \log(m/k)) \) bits; therefore we can encode our \( O(1/\tau) \) candidates using \( O((1/\tau) \log \log n) \) bits in total. Added over all the \( O(\log n) \) values of \( i \),\textsuperscript{5} the candidates \( C_i(x) \) require \( O((1/\tau) \log \log n) \) words per marked (i.e., branching) node. Added over all the branching nodes of \( F_i' \), this amounts to \( O(|F_i'|) \) space. The other pointers of \( F_i \), as well as node labels, can be represented normally, as they are \( O(n) \) in total.

\textsuperscript{5} The values of \( i \) are also bounded by \( O(\log((1/\tau) \log n)) \), but the bound \( O(\log n) = O(w) \) is more useful this time.
The small nodes left out from the trees \( F_i \) form a forest of sub-trees of size \( O((1/\tau) \log \log n) \) each. We can iterate this process \( \kappa \) times, so that the smallest trees are of size \( O((1/\tau) \log^{[\kappa]} n) \). We build no candidates sets on the smallest trees. We say that \( T' \) is a sub-tree of level 1, our \( F_i' \) are sub-trees of level 2, and so on, until the smallest sub-trees, which are of level \( \kappa \). Every node in \( T \) has a pointer to the root of the subtree where it belongs in the stratification.

The general process to solve a path \( \tau \)-majority query from \( u \) to \( v \) is then as follows. We compute \( z = \text{leaf}(u,v) \) and split the path from \( u \) to \( z \) into \( k-k'+1 \) sub-paths, where \( k' \) and \( k \) (note \( k' \leq k \leq \kappa \)) are the levels of the sub-tree where \( z \) and \( u \) belong, respectively. Let \( u_i \) be the root of the sub-tree of level \( i \) that is an ancestor of \( u \), except that we set \( u_k = z \).

1. If \( k = \kappa \), then \( u \) belongs to one of the smallest sub-trees. We then collect the \( O((1/\tau) \log^{[\kappa]} n) \) node weights in the path from \( u \) to \( u_\kappa \) one by one and include them in the set of candidates.
   Then we move to the parent of that root, setting \( u \leftarrow \text{parent}(u_\kappa) \) and \( k \leftarrow \kappa - 1 \).
2. At levels \( k' \leq k < \kappa \), if \( u \) is a branching node, we collect the \( 2/\tau \) candidates from the corresponding set \( C_i(u) \), where \( i \) is sufficient to cover \( u_k \) (\( C_i(u) \) will not store candidates beyond the sub-tree root). We then set \( u \leftarrow \text{parent}(u_k) \) and \( k \leftarrow k - 1 \).
3. At levels \( k' \leq k < \kappa \), if \( u \) is not a branching node, let \( x \) be lowest between \( \text{parent}(z) \) and the nearest branching ancestor of \( u \). Let also \( p \) be the position of \( u \) in \( S \). Then we find the \( 1/\tau \) \( \tau \)-majorities in \( S[p..p+\text{depth}(u)-\text{depth}(x)-1] \) in time \( O(1/\tau) \). We then continue from \( u \leftarrow x \) and \( k \leftarrow k(x) \), where \( k(x) \) is the level of the sub-tree where \( x \) belongs. Note that \( k(x) \) can be equal to \( k \), but it can also be any other level \( k' \leq k(x) < k \).
4. We stop when \( u = \text{parent}(z) \).

A similar procedure is followed to collect the candidates from \( v \) to \( z' \). In total, since each path has at most one case 2 and one case 3 per level \( k \), we collect at most \( 4\kappa \) candidate sets of size \( O(1/\tau) \), plus two of size \( O((1/\tau) \log^{[\kappa]} n) \). The total cost to verify all the candidates is then \( O((1/\tau)(\kappa + \log^{[\kappa]} n) \log \log n) \). The data structure uses linear space for any choice of \( \kappa \), whereas the optimal time is obtained by setting \( \kappa = \log^* n \).

The construction time, using the technique of Lemma 4 in level 1, is \( O(n \log \log n) \), since \( T' \) has \( t = O(n) \) nodes and \( m = O(\tau n/\log n) \) marked nodes. For higher levels, we use the basic quadratic method described in the first lines of Section 3.2: a subtree \( F \) of level \( k \) has \( t = O((1/\tau) \log^{[k-1]} n) \) nodes and \( m = O(\tau \tau/\log^{[k]} n) \) marked nodes, so it is built in time \( O(mt) \). There are \( O(\tau n/\log^{[k-1]} n) \) trees of level \( k \), which gives a total construction time of \( O(n \log^{[k-1]} n/\log^{[k]} n) \) for all the nodes in level \( k \). Added over all the levels \( k > 1 \), this yields \( O(n \log n/\log \log n) \). Both times, for \( k = 1 \) and \( k > 1 \), are however dominated by the \( O(n \log n) \) time to build the range majority data structure on \( S \).

\begin{theorem}
Let \( T \) be a tree of \( n \) nodes with labels in \([1..\sigma] \), and \( 0 < \tau < 1 \). On a RAM machine of \( w \)-bit words, we can build in \( O(n \log n) \) time an \( O(n) \) space data structure that answers path \( \tau \)-majority queries in time \( O((1/\tau) \log^* n \log \log n) \).
\end{theorem}

On the other hand, we can use any constant number of levels, and build the data structure of Section 3 on the last one, so as to ensure query time \( O(1/\tau) \) in this level as well. We use, however, the compressed storage of the candidates used in this section. With this storage format, a candidate set \( C_i(x) \) takes \( O((1/\tau) \log^{[\kappa]} n) \) bits. Multiplying by \( \log n \) (the crude upper bound on the number of \( i \) values), this becomes \( O((1/\tau) \log^{[\kappa]} n) \) words. Since the trees are of size \( O((1/\tau) \log^{[\kappa-1]} n) \) and the sampling rate used in Section 3 is \( \tau \), this amounts to \( O((1/\tau) \log^{[\kappa-1]} n \log^{[\kappa]} n) \) space per tree. Multiplied by the \( O(\tau n/\log^{[\kappa-1]} n) \) trees of level \( \kappa \), the total space is \( O(n \log^{[\kappa]} n) \).
The construction time of the candidate sets in the last level, using the basic quadratic construction, is $O(mt) = O((1/\tau)(\lg^{k-1} n)^2)$, because $t = O((1/\tau)\lg^{k-1} n)$ and $m = \tau t$ according to the sampling used in Section 3. Multiplying by the $O(\tau n/\lg^{k-1} n)$ trees of level $\kappa$, the total construction time for this last level is $O(n\lg^{k-1} n)$, again dominated by the time to build the range majority data structures if $\kappa > 1$. This yields the next result.

\textbf{Theorem 7.} Let $T$ be a tree of $n$ nodes with labels in $[1..\sigma]$, and $0 < \tau < 1$. On a RAM machine of $w$-bit words, for any constant $\kappa > 1$, we can build in $O(n\lg n)$ time an $O(n\lg^{[\kappa]} n)$ space data structure that answers path $\tau$-majority queries in time $O((1/\tau)\lg\lg w \sigma)$.

\section{A Succinct Space Solution}

The way to obtain a succinct space structure from Theorem 6 is to increase the thresholds that define the large nodes in Section 4. In level 1, we now define the large nodes as those whose subtree size is larger than $(1/\tau)(\lg n)^3$; in level 2, larger than $(1/\tau)(\lg\lg n)^3$; and in general in level $k$ as those with subtree size larger than $(1/\tau)(\lg^k n)^3$. This makes the space of all the $C_k(x)$ structures to be $o(n)$ bits. The price is that the traversal of the smallest trees now produces $O((1/\tau)(\lg^k n)^3)$ candidates, but this is easily sorted out by using $k + 1$ levels, since $(\lg^{k+1} n)^3 = o((\lg^k n)^3)$. To obtain succinct space, we will need that there are $o(n)$ subtrees of the smallest size, but that we find only $O((1/\tau)\lg^* n)$ candidates in total. Thus we set $\kappa = \lg^* n - \lg^* w$, so that there are $O(\kappa) = O(\lg^* n)$ levels, and the last-level subtrees are of size $O((1/\tau)(\lg^{k+1} n)^3) = O((1/\tau)\lg^{k-1} n) = o((1/\tau)\lg^* n)$. Still, there are $O(\tau n/(\lg^{k+1} n)^3) = O(\tau n/(\lg^{k-1} n)^3) = o(n)$ subtrees in the last level.

The topology of the whole tree $T$ can be represented using balanced parentheses in $2n + o(n)$ bits, supporting in constant time all the standard tree traversal operations we use \cite{17}. We assume that opening and closing parentheses are represented with 1s and 0s in $P$, respectively. Let us now focus on the less standard operations needed.

\subsection{Counting labels in paths}

In Section 3, we count the number of times a label $\ell$ occurs in the path from $u$ to the root by means of a query \texttt{labelanc} and by storing \texttt{count} fields in the nodes. In Section 4, we use in addition a string $S$ to support range majority queries on the unary paths.

To solve \texttt{labelanc} queries, we use the representation of Durocher et al. \cite[Lem. 7]{11}, which uses $nH + 2n + o(n)(H + 1)$ bits in addition to the $2n + o(n)$ bits of the tree topology. This representation includes a string $S[1..n]$ where all the labels of $T$ are written in preorder; any implementation of $S$ supporting \texttt{access}, \texttt{rank}, and \texttt{select} in time $O(\lg\lg w \sigma)$ can be used (e.g., \cite{4}). This string can also play the role of the one we call $S$ in Section 4: the labels of unary paths are contiguous in $S$, and any node $v$ can access its label from $S[\text{preorder}(v)]$.

On top of this string we must also answer range $\tau$-majority queries in time $O((1/\tau)\lg\lg w \sigma)$. We can use the slow variant of the succinct structure described in Section 2.3, which requires only $o(n)(H + 1)$ additional bits and also supports \texttt{access} in $O(1)$ time and \texttt{rank} and \texttt{select} in time $O(\lg\lg w \sigma)$. This variant of the structure is built in $O(n\lg n)$ time.

In addition to supporting operation \texttt{labelanc}, we need to store or compute the \texttt{count} fields. Durocher et al. \cite{11} also require this field, but find no succinct way to represent it. We now show how to obtain this value within succinct space.
The sequence $S$ lists the labels of $T$ in preorder, that is, aligned with the opening parentheses of $P$. Assume we have another sequence $S'[1..n]$ where the labels of $T$ are listed in postorder (i.e., aligned with the closing parentheses of $P$). Since the opened parentheses not yet closed in $P[1..i]$ are precisely node $i$ and its ancestors, we can compute the number of times a label $\ell$ appears in the path from $P[i]$ to the root as $\text{rank}_L(S, \text{rank}_1(P, i)) - \text{rank}_L(S', \text{rank}_0(P, i))$.

Therefore, we can support this operation with $nH + o(n)(H + 1)$ additional bits. Note that, with this representation, we do not need the operation $\text{labelanc}$, since we do not need that $P[i]$ itself is labeled $\ell$.

If we do use operation $\text{labelanc}$, however, we can ensure that $P[i]$ is labeled $\ell$, and another solution is possible based on partial rank queries. Let $o = \text{rank}_L(S, \text{rank}_1(P, i))$ and $c = \text{rank}_L(S', \text{rank}_0(P, i))$ be the numbers of opening and closing parentheses up to $P[i]$, so that we want to compute $o - c$. Since $P[i]$ is labeled $\ell$, it holds that $S[\text{rank}_1(P, i)] = \ell$, and thus $o = \text{prank}(S, \text{rank}_1(P, i))$. To compute $c$, we do not store $S'$, but rather $S''[1..2n]$, so that $S''[i]$ is the label of the node whose opening or closing parenthesis is at $P[i]$ (i.e., $S''$ is formed by interleaving $S$ and $S'$). Then, $\text{prank}(S'', i) = o + c$; therefore the answer we seek is $o - c = 2 \cdot \text{prank}(S, \text{rank}_1(P, i)) - \text{prank}(S'', i)$.

We use the structure for constant-time partial rank queries [3, Sec. 3] that requires $O(n) + o(nH)$ bits on top of a sequence that can be accessed in $O(1)$ time. We can build it on $S$ and also on $S''$, though we do not explicitly represent $S''$: any access to $S''$ is simulated in constant time with $S''[i] = S[\text{rank}_1(P, i)]$ if $P[i] = 1$, and $S''[i] = S[\text{rank}_1(P, \text{open}(P, i))]$ otherwise. This partial rank structure is built in $O(n)$ randomized time and in $O(n \log n)$ time w.h.p.\footnote{It involves building perfect hash functions, which succeeds with constant probability $p$ in time $O(n)$. Repeating $c \log n$ times, the failure probability is $1 - O(1/n^{c/\log(1/p)})$.}

### 5.2 Other data structures

The other fields stored at tree nodes, which we must now compute within succinct space, are the following:

**Pointers to candidate sets $C_i(x)$.** All the branching nodes in all subtrees except those of level $\kappa + 1$ are marked, and there are $O(n/(\log^{\kappa + 1} n)^3) = o(n)$ such nodes. We can then mark their preorder ranks with 1s in a bitvector $M[1..n]$. Since $M$ has $o(n)$ 1s, it can be represented within $o(n)$ bits [18] while supporting constant-time $\text{rank}$ and $\text{select}$ operations. We can then find out when a node $i$ is marked (iff $M[\text{preorder}(i)] = 1$), and if it is, its rank among all the marked nodes, $r = \text{rank}_1(M, \text{preorder}(i))$. The $C_i(x)$ sets of all the marked nodes $x$ of any level can be written down in a contiguous memory area of total size $o(n)$ bits, sorted by the preorder rank of $x$. A bitvector $C$ of length $o(n)$ marks the starting position of each new node $x$ in this memory area. Then the area for marked node $i$ starts at $p = \text{select}_1(C, r)$. A second bitvector $D$ can mark the starting position of each $C_j(x)$ in the memory area of each node $x$, and thus we access the specific set $C_j(x)$ from position $\text{select}_1(D, \text{rank}_1(D, p - 1) + j)$.

**Pointers to subtree roots.** We store an additional bitvector $B[1..2n]$, parallel to the parentheses bitvector $P[1..2n]$. In $B$, we mark with 1s the positions of the opening and closing parentheses that are roots of subtrees of any level. As there are $O(n/(\log^{\kappa + 1} n)^3) = o(n)$ such nodes, $B$ can be represented within $o(n)$ bits while supporting constant-time $\text{rank}$ and
select operations. We also store the sequence of $o(n)$ parentheses $P'$ corresponding to those in $P$ marked with a 1 in $B$. Then the nearest subtree root containing node $P'[i]$ is obtained by finding the nearest position to the left marked in $B$, $r = \text{rank}_1(B,i)$ and $j = \text{select}_1(B,r)$, and then considering the corresponding node $P'[r]$. If it is an opening parenthesis, then the nearest subtree root is the node whose parenthesis opens in $P'[j]$. Otherwise, it is the one opening at $P'[j']$, where $j' = \text{select}_1(B,\text{enclose}(P',\text{open}(P',r)))$ (see [19, Sec. 4.1]).

Finding the nearest branching ancestor. A unary path looks like a sequence of opening parentheses followed by a sequence of closing parentheses. The nearest branching ancestor of $P[i]$ can then be obtained in constant time by finding the nearest closing parenthesis to the left, $r = \text{select}_1(\text{rank}_1(P,i))$, and the nearest opening parenthesis to the right, $r = \text{select}_1(\text{close}(P,i)) + 1$. Then the answer is the larger between $\text{enclose}(P,\text{open}(P,l))$ and $\text{enclose}(P,r)$.

Determining the subtree level of a node. Since we can compute $s = \text{subtreesize}(i)$ of a node $P[i]$ in constant time, we can determine the corresponding level: if $s > (1/\tau) \lg^3 n$, it is level 1. Otherwise, we look up $\tau \cdot s$ in a precomputed table of size $O(\lg^3 n)$ that stores the level corresponding to each possible size.

Therefore, depending on whether we represent both $S$ and $S'$ or use partial rank structures, we obtain two results within succinct space.

Theorem 8. Let $T$ be a tree of $n$ nodes with labels in $[1..\sigma]$, and $0 < \tau < 1$. On a RAM machine of $w$-bit words, we can build in $O(n \lg n)$ time a data structure using $2nH + 4n + o(n)(H + 1)$ bits, where $H \leq \lg \sigma$ is the entropy of the distribution of the node labels, that answers path $\tau$-majority queries in time $O((1/\tau) \lg^2 n \lg \lg_w \sigma)$.

Theorem 9. Let $T$ be a tree of $n$ nodes with labels in $[1..\sigma]$, and $0 < \tau < 1$. On a RAM machine of $w$-bit words, we can build in $O(n \lg n)$ time (w.h.p.) a data structure using $nH + O(n) + o(nH)$ bits, where $H \leq \lg \sigma$ is the entropy of the distribution of the node labels, that answers path $\tau$-majority queries in time $O((1/\tau) \lg^* n \lg \lg_w \sigma)$.

We note that, within this space, all the typical tree navigation functionality, as well as access to labels, is supported.

6 Conclusions

We have presented the first data structures that can efficiently find the $\tau$-majorities on the path between any two given nodes in a tree. Our data structures use linear or near-linear space, and even succinct space, whereas our query times are close to optimal, by a factor near log-logarithmic.

As mentioned in the Introduction, many applications of these results require that the trees are multi-labeled, that is, each node holds several labels. We can easily accommodate multi-labeled trees $T$ in our data structure, by building a new tree $T'$ where each node $u$ of $T$ with $m(u)$ labels $\ell_1, \ldots, \ell_{m(u)}$ is replaced by an upward path of nodes $u_1, \ldots, u_{m(u)}$, each $u_i$ holding the label $\ell_i$ and being the only child of $u_{i+1}$ (and $u_{m(u)}$ being a child of $v_1$, where $v$ is the parent of $u$ in $T$). Path queries from $u$ to $v$ in $T$ are then transformed into path queries from $u_1$ to $v_1$ in $T'$, except when $u$ ($v$) is an ancestor of $v$ ($u$), in which case we replace $u$ ($v$) by $u_{\ell_i(m)}$ ($v_{\ell_i(m)}$) in the query. All our complexities then hold on $T'$, which is of size $n = |T'| = \sum_{u \in T} m(u)$.
Our query time for path $\tau$-majorities in linear space, $O((1/\tau) \lg^* n \log \log \sigma)$, is over the optimal time $O(1/\tau)$ that can be obtained for range $\tau$-majorities on sequences [1]. It is open whether we can obtain optimal time on trees within linear or near-linear space. Other interesting research problems are solving $\tau'$-majority queries for any $\tau' \geq \tau$ given at query time, in time proportional to $1/\tau'$ instead of $1/\tau$, and to support insertions and deletions of nodes in $T$. Similar questions can be posed for $\tau$-minorities, where the $O((1/\tau) \log \log \sigma)$ query time of our linear-space solutions is also over the time $O(1/\tau)$ achievable on sequences [1].

References

Encoding Two-Dimensional Range Top-$k$ Queries Revisited

Seungbum Jo
University of Siegen, Germany
Seungbum.Jo@uni-siegen.de
https://orcid.org/0000-0002-8644-3691

Srinivasa Rao Satti
Seoul National University, South Korea
ssrao@cse.snu.ac.kr
https://orcid.org/0000-0003-0636-9880

Abstract

We consider the problem of encoding two-dimensional arrays, whose elements come from a total order, for answering Top-$k$ queries. The aim is to obtain encodings that use space close to the information-theoretic lower bound, which can be constructed efficiently. For $2 \times n$ arrays, we first give upper and lower bounds on space for answering sorted and unsorted 3-sided Top-$k$ queries. For $m \times n$ arrays, with $m \leq n$ and $k \leq mn$, we obtain $(m \lg \binom{k+1}{n} + 4nm(m - 1) + o(n))$-bit encoding for answering sorted 4-sided Top-$k$ queries. This improves the $\min \{O(mn \lg n), m^2 \lg \binom{k+1}{n} + m \lg m + o(n)\}$-bit encoding of Jo et al. [CPM, 2016] when $m = o(\lg n)$. This is a consequence of a new encoding that encodes a $2 \times n$ array to support sorted 4-sided Top-$k$ queries on it using an additional $4n$ bits, in addition to the encodings to support the Top-$k$ queries on individual rows. This new encoding is a non-trivial generalization of the encoding of Jo et al. [CPM, 2016] that supports sorted 4-sided Top-$2$ queries on it using an additional $3n$ bits. We also give almost optimal space encodings for 3-sided Top-$k$ queries, and show lower bounds on encodings for 3-sided and 4-sided Top-$k$ queries.

2012 ACM Subject Classification Theory of computation → Data compression

Keywords and phrases Encoding model, top-$k$ query, range minimum query

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.69


Acknowledgements We would like to thank Moses Ganardi for helpful discussions

1 Introduction

Given a one-dimensional (1D) array $A[1...n]$ of $n$ elements from a total order, the range Top-$k$ query on $A$ (Top-$k(i, j, A), 1 \leq i, j \leq n$) returns the positions of $k$ largest values in $A[i...j]$. In this paper, we refer to these queries as 2-sided Top-$k$ queries; and the special case where the query range is $[1...i]$, for $1 \leq i \leq n$, as the 1-sided Top-$k$ queries. We can extend the definition to the two-dimensional (2D) case – given an $m \times n$ 2D array $A[1...m][1...n]$ of $mn$ elements from a total order and a $k \in \{1, \ldots, mn\}$, the range Top-$k$ query on $A$ (Top-$k(i, j, a, b, A), 1 \leq i, j \leq m, 1 \leq a, b \leq n$) returns the positions of $k$ largest values in

1 The author of this paper is supported by the DFG research project LO748/11-1.
Encoding Two-Dimensional Range Top-$k$ Queries Revisited

$A[i \ldots j][a \ldots b]$. Without loss of generality, we assume that all elements in $A$ are distinct (by ordering equal elements based on the lexicographic order of their positions). Also, we assume that $m \leq n$. In this paper, we consider the following types of Top-$k$ queries.

- Based on the order in which the answers are reported
  - Sorted query: the $k$ positions are reported in sorted order of their corresponding values.
  - Unsorted query: the $k$ positions are reported in an arbitrary order.
- Based on the query range
  - 3-sided query: the query range is $A[i \ldots j][1 \ldots b]$, for $i, j \in \{1, m\}$, and $b \in \{1, n\}$.
  - 4-sided query: the query range is $A[i \ldots j][a \ldots b]$, for $i, j \in \{1, m\}$, and $a, b \in \{1, n\}$.

We consider how to support these range Top-$k$ queries on $A$ in the encoding model. In this model, one needs to construct a data structure (an encoding) so that queries can be answered without accessing the original input array $A$. The minimum size of an encoding is also referred to as the effective entropy of the input data [8]. Our aim is to obtain encodings that use space close to the effective entropy, which can be constructed efficiently. In the rest of the paper, we use Top-$k(i, j, a, b)$ to denote Top-$k(i, j, a, b, A)$ if $A$ is clear from the context. Also, unless otherwise mentioned, we assume that all Top-$k$ queries are sorted 4-sided Top-$k$ queries. Finally, we assume the standard word-RAM model [14] with word size $\Theta(\lg n)$.

1.1 Previous work

The problem of encoding 1D and 2D arrays to support Top-$k$ queries has been widely studied in the recent years. Especially, the case when $k = 1$, which is commonly known as the Range maximum query (RMQ) problem, has been studied extensively, and has a wide range of applications [1]. Optimal encodings for answering RMQ queries on 1D and 2D arrays are well-studied. Fischer and Heun [5] proposed a $2n + o(n)$-bit data structure which answers RMQ queries on 1D array of size $n$ in constant time. For a 2D array $A$ of size $m \times n$, a trivial way to encode $A$ for answering RMQ queries is to store the rank of all elements in $A$, using $O(nm \lg n)$ bits. Golin et al. [8] show that when $m = 2$ and RMQ encodings on each row are given, one can support RMQ queries on $A$ using $n - O(\lg n)$ extra bits by encoding joint Cartesian tree on both rows. By extending the above encoding, they obtained $nm(m + 3)/2$-bit encoding for answering RMQ queries on $A$, which takes less space than the trivial $O(nm \lg n)$-bit encoding when $m = o(\lg n)$. Brodal et al. [3] proposed an $O(\min(nm \lg n, m^2n))$-bit data structure which supports RMQ queries on $A$ in constant time. Finally, Brodal et al. [2] obtained an optimal $O(nm \lg m)$-bit encoding for answering RMQ queries on $A$ (although the queries are not supported efficiently).

For the case when $k = 2$, Davoodi et al. [4] proposed a $3.272n + o(n)$-bit data structure to encode a 1D array of size $n$, which supports Top-2 queries in constant time. The space was later improved by Gawrychowski and Nicholson [7] to the optimal $2.755n + o(n)$ bits, although it does not support queries efficiently. For Top-2 queries on $2 \times n$ array $A$, Jo et al. [11] showed that $3n + o(n)$-bit extra space is enough for answering 4-sided Top-2 queries on $A$, when encodings of 2-sided Top-2 queries for each row are given.

For general $k$, on a 1D array of size $n$, Grossi et al. [10] proposed an $O(n \lg k)$-bit encoding which supports sorted Top-$k$ queries in $O(k)$ time, and showed that at least $n \lg k - O(n)$ bits are necessary for answering 1-sided Top-$k$ queries; Gawrychowski and Nicholson [7] proposed a $(k + 1)nH(1/(k + 1)) + o(n)$-bit encoding for Top-$k$ queries (although the queries are not supported efficiently).

---

2 We use $\lg n$ to denote $\log_2 n$.

3 $H(x) = x \lg (1/x) + (1 - x) \lg (1/(1 - x))$, i.e., an entropy of the binary string whose density of zero is $x$. 

Table 1 Summary of the results of upper and lower bounds for Top-k encodings on 2D arrays. The lower bound results marked (*) (of Theorem 12 and 13) are for the additional space (in bits) necessary, assuming that encodings of Top-k queries for both rows are given.

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Query type</th>
<th>Space (in bits)</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Upper bounds</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 × n</td>
<td>3-sided, unsorted</td>
<td>$2 \log \left(\frac{(k+1)n}{n}\right) + \left[(n - \lfloor k/2 \rfloor) \log 3\right] + o(n)$</td>
<td>Theorem 2</td>
</tr>
<tr>
<td></td>
<td>3-sided, sorted</td>
<td>$2 \log \left(\frac{m n}{n}\right) + 2n + o(n)$</td>
<td>Theorem 3, $k = 2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2 \log \left(\frac{(k+1)n}{n}\right) + \left[2n \log 3\right] + o(n)$</td>
<td>Theorem 4</td>
</tr>
<tr>
<td></td>
<td>4-sided, sorted</td>
<td>$\frac{5n - O(\log n)}{m}$</td>
<td>[8], $k = 1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2 \log \left(\frac{m n}{n}\right) + 3n + o(n)$</td>
<td>[11], $k = 2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$2 \log \left(\frac{(k+1)n}{n}\right) + 4n + o(n)$</td>
<td>Theorem 8</td>
</tr>
<tr>
<td>m × n</td>
<td>4-sided, sorted</td>
<td>$O(\min (nm \log n, m^2 n))$</td>
<td>[8], $k = 1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(nm \log m)$</td>
<td>[2], $k = 1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$m^2 \log \left(\frac{(k+1)n}{n}\right) + m \log m + o(n)$</td>
<td>[11]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$m \log \left(\frac{(k+1)n}{n}\right) + 2m(m - 1) + o(n)$</td>
<td>Theorem 9</td>
</tr>
<tr>
<td></td>
<td>Lower bounds</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 × n</td>
<td>4-sided</td>
<td>$\frac{5n - O(\log n)}{m}$</td>
<td>[8], $k = 1$</td>
</tr>
<tr>
<td></td>
<td>3-sided, unsorted</td>
<td>$\frac{1.27(n - k/2) - o(n)}{m}$</td>
<td>(*) Theorem 12</td>
</tr>
<tr>
<td></td>
<td>3 or 4-sided, sorted</td>
<td>$2n - O(\log n)$</td>
<td>(*) Theorem 13</td>
</tr>
<tr>
<td>m × n</td>
<td>4-sided, sorted</td>
<td>$\Omega(nm \log (\max (m, k)))$</td>
<td>[3, 10]</td>
</tr>
</tbody>
</table>

supported efficiently), and showed that at least $(k+1)nH(1/(k+1))(1-o(1))$ bits are required to encode Top-k queries. They also proposed a $(k+1.5)nH(1.5/(k+1.5)) + o(n \log k)$-bit data structure for answering Top-k queries in $O(k^6 \log^2 n f(n))$ time, for any strictly increasing function $f$. For a 2D array $A$ of size $m \times n$, one can answer Top-k queries using $O(nm \log n)$ bits, by storing the rank of all elements in $A$. Jo et al. [11] recently developed the first non-trivial Top-k encodings on 2D arrays. They proposed an $(m^2 \log \left(\frac{(k+1)n}{n}\right) + m \log m + o(n))$-bit encoding for sorted 4-sided Top-k queries, which takes less space than trivial $O(nm \log n)$-bit encoding when $n = \Omega(km)$. They also proposed an $O(nm \log n)$-bit data structure which supports Top-k queries in $O(k)$ time.

1.2 Our results

For any $2 \times n$ array $A$, we first show, in Section 2, that given the sorted 1-sided Top-k encodings of the two individual rows, we can support the 3-sided sorted (resp., unsorted) Top-k queries on $A$ using an additional $\left[(n - \lfloor k/2 \rfloor) \log 3\right] + o(n)$ (resp., $\left[2n \log 3\right] + o(n))$ bits. For unsorted queries, our encoding can answer the queries in $2T(n, k) + O(1)$ time, when one can answer the 1-sided sorted Top-k queries for each row in $T(n, k)$ time.

For 4-sided Top-k queries on $A$, we show that $4n$ bits are sufficient for answering sorted 4-sided Top-k queries on $2 \times n$ array, when encodings for answering sorted 2-sided Top-k queries for each row are given. This encoding is obtained by extending a DAG for answering Top-2 queries on $2 \times n$ array which is proposed by Jo et al. [11], but we use a different approach from their encoding to encode the DAG. Our result generalizes the $(5n - O(\log n))$-bit encoding of RMQ query on $2 \times n$ array proposed by Golin et al. [8] to general $k$, and shows that we can encode a joint Cartesian tree for general $k$ (which corresponds to the DAG in our paper) using $4n$ bits. Note that the additional space is independent of $k$. We also obtain a data structure for answering Top-k queries in $O(k^2 + kT(n, k))$ time using $2S(n, k) + (4k + 7)n + ko(n)$ bits,
if there exists an $S(n,k)$-bit encoding to answer sorted 2-sided Top-$k$ queries on a 1D array of size $n$ in $T(n,k)$ time. Comparing to the $2S(n,k)+4n+o(n)$-bit encoding, this data structure uses more space but supports Top-$k$ queries efficiently (the $2S(n,k)+4n+o(n)$-bit encoding takes $O(k^2n^2+nkT(n,k))$ time for answering Top-$k$ queries).

By extending the $2S(n,k)+4n+o(n)$-bit encoding on $2 \times n$ array, we obtain $(m \lg \left(\frac{2k}{n}\right) + 2nm(m-1)+o(n))$-bit encoding for answering 4-sided Top-$k$ queries on $m \times n$ arrays. This improves upon the trivial $O(mn \lg n)$-bit encoding when $m = o(\lg n)$, and also generalizes the $nm(m+3)/2$-bit encoding [8] for answering RMQ queries. Comparing with Jo et al.’s [11] $(m^2 \lg \left(\frac{2k}{n}\right) + mlg m + o(n))$-bit encoding, our encoding takes less space in all cases (for $k > 1$) since $m^2 \lg \frac{2k}{n} = mlg \frac{2k}{n} + m(m-1) \lg \frac{2k}{n}$. The trivial encoding of the input array takes $O(mn \lg n)$ bits, whereas one can easily show a lower bound of $\Omega(mn \lg (\max(m,k)))$ bits for any encoding of an $m \times n$ array that supports Top-$k$ queries since at least $O(mn \lg m)$ bits are necessary for answering RMQ queries [3], and at least $nlg k$ bits are necessary for answering Top-$k$ queries for each row [10]. Thus, there is only a small range of parameters where a strict improvement over the trivial encoding is possible. Our result closes this gap partially, achieving a strict improvement when $m = o(\lg n)$.

Finally in Section 4, given a $2 \times n$ array $A$, we consider the lower bound on additional space required to answer unsorted (or sorted) Top-$k$ on $A$ when encodings of Top-$k$ query for each row are given. We show that at least $1.27(n-k)/2 - o(n)$ (or $2n - O(\lg n)$) additional bits are necessary for answering unsorted (or sorted) 3-sided Top-$k$ queries on $A$, when encodings of unsorted (or sorted) 1-sided Top-$k$ query for each row are given. We also show that $2n - O(\lg n)$ additional bits are necessary for answering sorted 4-sided Top-$k$ queries on $A$, when encodings of unsorted (or sorted) 2-sided Top-$k$ query for each row are given. These lower bound results imply that our encodings in Sections 2 and 3 are close to optimal (i.e., within $O(n)$ bits of the lower bound), since any Top-$k$ encoding for the array $A$ also needs to support the Top-$k$ queries on the individual rows. All these results are summarized in Table 1.

### 2 Encoding 3-sided range Top-$k$ queries on $2 \times n$ array

In this section, we consider the upper bounds on space for encoding unsorted and sorted 3-sided Top-$k$ queries on $2 \times n$ array $A[1,2][1 \ldots n]$, given the encodings of Top-$k$ on the two individual rows. For the case of $k = 1$ (i.e., the RMQ problem), there exists an optimal $(5n - O(\lg n))$-bit encoding of a $2 \times n$ array, which stores two Cartesian trees for the individual rows, and encodes the additional information (to answer the queries involving both rows) using a joint Cartesian tree [8]. In the rest of this section, we assume that $k > 1$. We first consider answering unsorted and sorted 3-sided Top-$k$ queries. If sorted 1-sided Top-$k$ queries on each row can be answered using $S(n,k)$ space\(^4\), we can support unsorted and sorted 3-sided Top-$k$ queries on $A$ using $(2S(n,k) + [(n-k)/2] \lg 3))$ and $(2S(n,k) + [2n \lg 3] + o(n))$ bits respectively. For $i \in \{1,2\}$, let $A_i = [a_{i1}, \ldots, a_{in}]$ be an array of size $n$ constituting the $i$-th row of $A$ and let $(i,j)$ denote the position in the $i$-th row and $j$-th column in $A$. We first introduce a lemma from Grossi et al. [9], to support queries efficiently.

\(^4\) here and in the rest of the paper, we assume that $S(n,k) = S(n,n)$ for $k > n$
Lemma 1 ([9]). Let $A$ be an array of size $n$ over an alphabet of size 3. Then one can encode $A$ using at most $nH_0(A) + o(n) = \lfloor n \log 3 \rfloor + o(n)$ bits, while supporting the following queries in $O(1)$ time ($H_0(A)$ denotes the zeroth-order entropy of $A$).

- $\text{rank}_A(x, i) :$ returns the number of occurrence of the symbol $x$ in $A[1 \ldots i]$
- $\text{select}_A(x, i) :$ returns the position of the $i$-th occurrence of the symbol $x$ in $A$.

Also, we define $\text{Top-}^3_{k}(x)$ which can answer the sorted 1-sided $1 \times n$ query, we can answer the query in $O(n)$ time. Also if one can construct another encoding that uses $2n \log(n) \log \log(n) + o(n)$ bits.

Encoding unsorted 3-sided Top-$k$ queries on $2 \times n$ array. We now show how to support (unsorted and sorted) 3-sided Top-$k$ queries on a $2 \times n$ array $A$, given the sorted 1-sided Top-$k$ encodings on the two rows $A_1$ and $A_2$. (Note that in 1D, the space used by the sorted and unsorted 1-sided Top-$k$ encodings differ by $O(k \log k)$ bits.) For $1 \leq i \leq n$, let $f_i$ and $s_i = k - f_i$ be the number of answers to the (sorted or unsorted) Top-$k(1, 2, 1, i)$ query that belong to the first row and the second row, respectively. We first consider the unsorted case. Since the encodings for answering unsorted 1-sided Top-$k$ queries on $A_1$ and $A_2$ are given, it is enough to show how to answer 1-sided Top-$k$ queries on $A$ (to support all possible unsorted 3-sided Top-$k$ queries).

Theorem 2. Let $A$ be a $2 \times n$ array. For $1 \leq k \leq 2n$, if we have $S(n, k)$-bit encoding which can answer the sorted 1-sided Top-$k$ queries for each row in $T(n, k)$ time, then we can answer unsorted 3-sided Top-$k$ queries on $A$ using $(2S(n, k) + [(n - [k/2]) \log 3] + o(n))$ bits with $2T(n, k) + O(1)$ query time.

Encoding sorted 3-sided Top-$k$ queries on $2 \times n$ array. We now consider the encoding for answering sorted 3-sided Top-$k$ queries on a $2 \times n$ array $A$, when sorted 1-sided Top-$k$ encodings for the two rows $A_1$ and $A_2$ are given. Similar to the unsorted case, it is enough to show how to support the sorted 1-sided Top-$k$ queries on $A$. We first give an encoding that uses less space for small values of $k$, and later give another encoding that is space-efficient for large values of $k$.

Theorem 3. Let $A$ be a $2 \times n$ array. For $1 \leq k \leq n$, if we have $S(n, k)$-bit encoding which can answer the sorted 1-sided Top-$k$ queries for each row in $T(n, k)$ time, then we can encode $A$ using $2S(n, k) + kn$ bits to support sorted 3-sided Top-$k$ queries in $2T(n, k)$ time.

The additional space used in Theorem 3 is close to the optimal for $k = 1, 2$ or 3, but increases with $k$. Using similar ideas, one can obtain another encoding that uses $2n \log(k + 1)$ bits, in addition to the individual row encodings. In the following, we give an alternative encoding whose additional space is independent of $k$.

Theorem 4. Let $A$ be a $2 \times n$ array. For $1 \leq k \leq 2n$, suppose we have $S(n, k)$-bit encoding which can answer the sorted 1-sided Top-$k$ queries. Then we can answer sorted 3-sided Top-$k$ queries on $A$ using $(2S(n, k) + [2n \log 3] + o(n))$ bits.

If we use the $(n \log k + O(n))$-bit Top-$k$ encoding of a 1D array by Grossi et al. [10] that can answer sorted 1-sided Top-$k$ query on 1D array of size $n$ in $O(k \log k)$ time, then we obtain 3-sided unsorted (or sorted) Top-$k$ encodings on $A$ using $2n \log k + O(n)$ bits. Furthermore for unsorted queries, we can answer the query in $O(k \log k)$ time. Also if one can construct an encoding for answering 1-sided unsorted (or sorted) Top-$k$ queries on individual rows...
in \(C(n, k)\) time, we can construct an encoding for answering 3-sided unsorted (or sorted) Top-\(k\) queries in \(O(C(n, k) + n \lg k)\) time as follows. Since it is enough to know the answers of unsorted (or sorted) Top-\(k\) queries for \(1 \leq i \leq n\) to construct, we maintain a min-heap and insert the first \(k\) values (in column-major order) to the heap and sort them using \(O(k \lg k)\) time. After that, whenever we insert a next value in \(A\) in column-major order, we delete the smallest value in the heap, using \(O((n - k) \lg k)\) time in total. The data structure of Lemma 1 can be constructed in \(O(n)\) time [9].

3 Encoding 4-sided Top-\(k\) queries on \(2 \times n\) array

In this section, we describe the encoding of sorted 4-sided Top-\(k\) on \(2 \times n\) array, assuming that sorted 2-sided Top-\(k\) encodings on \(A_1\) and \(A_2\) are given. We show that we can encode sorted 4-sided Top-\(k\) queries on \(A\) using at most \(2S(n, k) + 4n\) bits if sorted 2-sided Top-\(k\) queries on each row can be answered in \(T(n, k)\) time using \(S(n, k)\) bits. By extending this encoding to an \(n \times m\) array, we obtain an \(mS(n, k) + 2nm(m - 1)\)-bit encoding for answering the 4-sided Top-\(k\) query on \(m \times n\) array. Note that if we use Gawrychowski and Nicholson’s \((\lg \left(\binom{k+1}{n}\right) + o(n))\)-bit optimal encoding for sorted 2-sided Top-\(k\) queries on a 1D array [7], we obtain an encoding that takes \((m \lg \left(\binom{k+1}{n}\right) + 2nm(m - 1) + o(n))\) bits for answering 4-sided Top-\(k\) queries. This improves upon the trivial \(O(mn \lg n)\)-bit encoding when \(m = \alpha(\lg n)\), and comparing with Jo et al.’s [11] \((m^2 \lg \left(\binom{k+1}{n}\right) + m \lg m + o(n))\)-bit encoding, our encoding takes less space than in all cases when \(k > 1\). Finally for \(2 \times n\) array, we describe a data structure for answering Top-\(k\) queries in \(O(k^2 + kT(n, k))\) time using \(2S(n, k) + (4k + 7)n + ko(n)\) bits, which supports Top-\(k\) queries in efficient time, and for small constant \(k\) \((2 \leq k < 160)\), this data structure takes less space than constructing a data structure of Grossi et al. [10] on the array of size \(2n\) which stores the values in \(A\) in column-major order.

We first define a binary DAG \(D_A^k\) on \(A\), which generalizes the binary DAG defined by Jo et al. to answer Top-2 queries on \(A\) [11]. Then we show how to encode \(D_A^k\) using \(4n\) bits, to answer the sorted 4-sided Top-\(k\) queries on \(A\). Every node \(p\) in \(D_A^k\) is labeled with some closed interval \(p = [a, b]\), where \(1 \leq a, b \leq n\). We use Top-\(k\) queries to refer to the sorted Top-\(k\) query \((1, 2, a, b, A)\). For a node \(p = [a, b]\) in \(D_A^k\) and \(1 \leq i \leq k\), let \((p^c_i, p^r_i)\) be the position of the \(i\)-th largest element in \(A[1, 2][a \ldots b]\). Now we define \(D_A^k\) as follows (see Figure 1 for an example.).

1. The root of \(D_A^k\) is labeled with the range \([1, n]\).
2. A node \([a, b]\) does not have any child node (i.e, leaf node) if \(2(b - a + 1) \leq k\).
3. Suppose there exists a non-leaf node \(p = [a, b]\) in \(D_A^k\), and let \(a' \leq b' \leq b\), where \(a \leq a' \leq b' \leq b\), be the leftmost and rightmost column indices among the answers of Top-\(k\) queries, respectively. If \(a < b'\), then the node \(p\) has a node \([a, b' - 1]\) as a left child. Similarly, if \(a' < b\), the node \(p\) has a node \([a' + 1, b]\) as a right child.

The following lemma states some useful properties of \(D_A^k\). All the statements in the lemma can be proved by simply extending the proofs of the lemmas in [11].

Lemma 5 ([11]). Let \(A\) be a \(2 \times n\) array. For any two distinct nodes \(p = [a_p, b_p]\) and \(q = [a_q, b_q]\) in \(D_A^k\), following statements hold.

(i) Top-\(k\) \((p) \neq \text{Top-} k\) \((q)\) (i.e., any two distinct nodes have different Top-\(k\) answers).

(ii) \(p \subset q\) if and only if \(p\) is descendant of \(q\).

(iii) For any interval \([a, b]\) with \(1 \leq a \leq b \leq n\), there exists a unique node \(p_{[a, b]}\) in \(D_A^k\) such that \([a, b] \subset p_{[a, b]}\), and any descendant of \(p_{[a, b]}\) does not contain \([a, b]\). Furthermore, for such a node \(p_{[a, b]}\), Top-\(k\) \(([a, b]) = \text{Top-} k\) \((p_{[a, b]})\).
By Lemma 5(iii), if the DAG $D_A^3$ and the answers for each sorted 2-sided $\text{Top-k}$ queries corresponding to all the nodes in $D_A^3$ are given, then we can answer any sorted $\text{Top-k}(1, 2, a, b, A)$ query by finding the corresponding node in $p_{[a,b]}$ in $D_A^3$.

Now we describe how to encode $D_A^k$ to answer the $\text{Top-k}(p)$ query for each node $p \in D_A^k$. Our encoding of $D_A^k$ uses a different approach from the encoding of Jo et al. [11], which encodes by traversing $D_A^3$ in level order. We say that a node $p = [a, b]$ picks the position $(x, y)$ if we store the information that $(x, y)$ is the position of the $i$-th largest element in $A[1, 2][a \ldots b]$, for some $i \leq k$. To encode the DAG $D_A^3$, we traverse its nodes in a modified level order, which we describe later. While traversing the nodes of $D_A^3$ in the modified level order, we classify the nodes as visited, half-visited, or unvisited. All the nodes are initially unvisited, and the traversal continues until all the nodes in $D_A^k$ are visited. During the traversal of unvisited or half-visited node, we may pick a position whose column index is contained in that node (under some conditions, described later). Whenever we pick a position, we store one bit of information to resolve some of the queries. We bound the overall additional space to $4n$ bits by showing that each position in $A$ is picked at most twice. For two nodes $p_i = [a_i, b_i]$ and $p_j = [a_j, b_j]$ with $p_i \not\subset p_j$ and $p_j \not\subset p_i$, we say the node $p_i$ precedes the node $p_j$ if $a_i < a_j$.

When the traversal starts at the root node $[1, n]$, we pick all positions which are answers to the $\text{Top-k}(1, n, A)$ query. Since we know the answers to the $\text{Top-k}(1, n, A_1)$ and $\text{Top-k}(1, n, A_2)$ queries, the positions that are picked at the root can be encoded using a $k$-bit sequence $a_1 \ldots a_k$ where $a_i$ represents the row index of the $i$-th largest element in $A[1, 2][1 \ldots n]$, for $1 \leq i \leq k$. From the definition of $D_A^3$, if the label of a node $p$ and the answers of the $\text{Top-k}(p)$ query are given, then it is easy to compute the labels of the children of $p$.

Since it is trivial to answer the $\text{Top-k}$ query at a leaf node, we only focus on the non-leaf nodes. Suppose that we traverse to a non-leaf node $p = [a, b]$, and let $q$ be one of its parent nodes (note that a node can have multiple parents in a DAG). Note that $1 \leq |\text{Top-k}(q) - \text{Top-k}(p)| \leq 2$, since $p$ contains all the $\text{Top-k}$ answers of $q$ except one or both positions from the column $a - 1$ or from the column $b + 1$. We first consider the case when $|\text{Top-k}(q) - \text{Top-k}(p)| = 1$ (this also includes the case when there exists another parent node $q'$ of $p$ such that $|\text{Top-k}(q) - \text{Top-k}(p)| = 1$ and $|\text{Top-k}(q') - \text{Top-k}(p)| = 2$). In this case, traversal visits the node $p$ only once in modified level order, and picks at most one position.

![Figure 1](2x_n_array_A_and_the_DAG_D_A^3.png)

**Figure 1** $2 \times n$ array $A$ and the DAG $D_A^3$. 

<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$A_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>17</td>
<td>12</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
</tr>
<tr>
<td>18</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>21</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
</tr>
</tbody>
</table>
Encoding Two-Dimensional Range Top-$k$ Queries Revisited

at node $p$. Let $\text{Top-}k(q) - \text{Top-}k(p) = \{(q_r^{k'}, q_s^{k'})\}$ for some $k' \leq k$. From the construction of $D_A^k$ and Lemma 5(ii), it is clear that $(p_r^{k'}, p_s^{k'}) = (q_r^{k'}, q_s^{k'})$ if $\ell < k'$; $(p_r^{k'}, p_s^{k'}) = (q_r^{k'+1}, q_s^{k'+1})$ if $k' \leq \ell < k$; and $\text{Top-}k(p) - \text{Top-}k(q) = \{(p_r^{k}, p_s^{k})\}$. Therefore, if the answers of the $\text{Top-}(k-1)(p)$ query are composed of $f_p$ positions from the first row and $s_p = (k - 1 - f_p)$ positions from the second row, then we can find $(p_r^{k}, p_s^{k})$ by comparing $(f_p + 1)$-th largest element in $A_1$ and $(s_p + 1)$-th largest element in $A_2$ using $\text{Top-}k(a, b, A_1)$ and $\text{Top-}k(a, b, A_2)$ queries, (we define the position of these elements as the first-candidates of node $p$), and choosing the position with larger element. Note that if the answers of the $\text{Top-}(k-1)(p)$ query contains all positions in $A[1][a \ldots b]$ or $A[2][a \ldots b]$, there is no first-candidate of node $p$ at the first or second row respectively. In this case we do not pick any positions at node $p$.

Now suppose that $(1, x)$ and $(2, y)$ are the first-candidates of node $p$, and without loss of generality suppose $A[1][x] > A[2][y]$, and hence $(p_r^{k}, p_s^{k}) = (1, x)$. Then we consider the following cases.

1. If both $(1, x)$ and $(2, y)$ is not picked in the former nodes in $D_A^k$ in modified level order, we pick $(1, x)$.

2. Suppose $(1, x)$ or $(2, y)$ is already picked by a visited or half-visited node $p' = [a', b']$.

Then we pick $(1, x)$ at node $p$ if and only if for all such $p'$ does not contain both $x$ and $y$.

Suppose that $\text{Top-}(k-1)(p)$ is given and one of the first-candidates is picked at node $p$. Then we can store its information using one bit, by representing the row index of the first-candidate picked at $p$.

Now consider the case $|\text{Top-}k(q) - \text{Top-}k(p)| = 2$, and let $\text{Top-}k(q) - \text{Top-}k(p) = \{(q_r^{k'}, q_s^{k'}), (q_r^{k''}, q_s^{k''})\}$ for some $k' < k'' \leq k$. In this case, the traversal visits the node $p$ twice in modified level order, and picks at most two positions at node $p$. From the construction of $D_A^k$ and Lemma 5(ii), it is clear that $(p_r^{k'}, p_s^{k'}) = (q_r^{k'}, q_s^{k'})$ if $\ell < k'$; $(p_r^{k'}, p_s^{k'}) = (q_r^{k'+1}, q_s^{k'+1})$ if $k' \leq \ell < k''$; $(p_r^{k''}, p_s^{k''}) = (q_r^{k''+1}, q_s^{k''+1})$ if $k'' \leq \ell < k - 1$; and $\text{Top-}k(p) - \text{Top-}k(q) = \{(p_r^{k-1}, p_s^{k-1}), (p_r^{k'}, p_s^{k'})\}$. Therefore, if the answers of $\text{Top-}(k-2)(p)$ query are composed of $f_p$ and $s_p = (k - 2 - f_p)$ positions in the first and the second row respectively, we can find $(p_r^{k-1}, p_s^{k-1})$ by comparing $(f_p + 1)$-th largest element in $A_1$ and $(s_p + 1)$ in $A_2$ using $\text{Top-}k(a, b, A_1)$ and $\text{Top-}k(a, b, A_2)$ query (we again define the position of these elements as the first-candidates of node $p$). Suppose that $(1, x)$ and $(2, y)$ are first-candidates of node $p$, and without loss of generality suppose $A[1][x] > A[2][y]$, and hence $(p_r^{k-1}, p_s^{k-1}) = (1, x)$. In this case, we first pick $(1, x)$ or do not pick anything at node $p$ by the procedure described above, when we first traverse $p$ in modified level order. When we visit $p$ for the second time, we can find $(p_r^{k'}, p_s^{k'})$ by comparing $A_2[y]$ with the $(f_p + 2)$-th largest element in $A_1$ (we define the positions of these elements as the second-candidates of node $p$), and choose the position with the larger element. Note that if the answers of the $\text{Top-}(k-1)(p)$ query contains all positions in $A[1][a \ldots b]$ or $A[2][a \ldots b]$, there is no second-candidate of node $p$ at the first or second row respectively. In this case we do not pick any positions at node $p$ during the second visit of $p$. Again, suppose that $(1, x')$ and $(2, y)$ are the second-candidates of node $p$ and without loss of generality suppose $A[1][x'] < A[2][y]$, and hence $(p_r^{k'}, p_s^{k'}) = (2, y)$. Then we consider the following cases.

1. If both $(1, x')$ and $(2, y)$ is not picked in the former nodes in $D_A^k$ in modified level order, we pick $(2, y)$.

2. Suppose $(1, x')$ or $(2, y)$ is already picked by the visited or half-visited $p'' = [a'', b'']$. Then we pick $(2, y)$ at node $p$ if and only if for all such $p''$ does not contain both $x'$ and $y$. 
Note that if $\text{Top-}(k-2)(p)$ is given and $(1,x)$ is picked at node $p$, we can store its information using one bit, by representing a row index of first-candidate picked at $p$. Similarly, if $\text{Top-}(k-1)(p)$ is given and $(2,y)$ is picked at node $p$, we can store its information using one more bit.

Now we describe the algorithm to traverse the nodes in $D_A^k$ in the modified level order. In modified level order, for any two nodes $p = [i,j]$ and $p' = [i',j']$, we traverse $p$ prior to $p'$ if and only if all column indices of $p'$'s first or second candidates are contained in $p$. Furthermore by the procedure described above, we do not pick any position at $p'$ in this case if there exists a position which is the first or second candidate of both $p$ and $p'$. In the DAG, the level of the node $p$, denoted by $l(p)$, is defined as the number of edges in the longest path from root to $p$.

1. Mark the root of $D_A^k$ as visited, and add its children into visit-list, which is an ordered list such that for two nodes $p$ and $q$ in visit-list, $p$ comes before $q$ in visit-list if and only if $l(p) < l(q)$ or $l(p) = l(q)$ and $p$ precedes $q$ in the DAG.
2. Find the leftmost unvisited or half-visited node $p$ from visit-list which satisfies one of the following conditions (without loss of generality, assume that $x \leq y$).
   - Number of first or second candidates of $p$ is less than 2.
   - First or second candidates of $p$ are $(1,x)$ and $(2,y)$, and there exists no node $p'$ in visit-list such that (a) $p \subset p'$, or (b) $p'$ precedes $p$ and $x \in p'$, or (c) $p$ precedes $p'$ and $y \in p'$.
   - Then we continue the traversal from $p$.
3. Let $q$ be a parent of $p$. If (i) $|\text{Top-k}(q) - \text{Top-k}(p)| = 1$, or (ii) $|\text{Top-k}(q) - \text{Top-k}(p)| = 2$ and $p$ is half-visited, or (iii) the number of first or second candidates of $p$ is less than 2, then mark $p$ as visited, delete $p$ from the visit-list, and insert $p$’s children (if any) to visit-list. If none of these three conditions hold, then mark $p$ as half-visited.
4. Repeat Steps 2 and 3 until all the nodes in $D_A^k$ are marked as visited.

For example we traverse the nodes of $D_A^k$ in Figure 1 as: $[1,9] \rightarrow [1,4] \rightarrow [1,4] \rightarrow [3,9] \rightarrow [1,2] \rightarrow [1,2] \rightarrow [3,6] \rightarrow [6,9] \rightarrow [6,9] \rightarrow [1,1] \rightarrow [2,2] \rightarrow [3,4] \rightarrow [4,6] \rightarrow [6,7] \rightarrow [8,9] \rightarrow [8,9] \rightarrow [3,3] \rightarrow [4,4] \rightarrow [5,6] \rightarrow [8,8] \rightarrow [9,9] \rightarrow [5,5] \rightarrow [6,6]$. During the traversal (in the above order), the position(s) picked at each node are: $\{(1,2),(1,5),(2,5)\} \rightarrow (1,3) \rightarrow (2,3) \rightarrow (2,7) \rightarrow (1,1) \rightarrow (1,2) \rightarrow \epsilon \rightarrow (1,7) \rightarrow (1,8) \rightarrow \epsilon \rightarrow \epsilon \rightarrow (2,4) \rightarrow \epsilon \rightarrow (1,6) \rightarrow (1,9) \rightarrow (2,9) \rightarrow \epsilon \rightarrow \epsilon \rightarrow \epsilon \rightarrow \epsilon \rightarrow \epsilon \rightarrow \epsilon$, respectively ($\epsilon$ indicates that no position is picked). Now we bound the total number of picked positions during the traversal of $D_A^k$.

**Lemma 6.** (+) Given $2 \times n$ array $A[1,2][1 \ldots n]$ and DAG $D_A^k$, any position in $A$ is picked at most twice while we traverse all nodes in $D_A^k$ in the modified level order.

Now we prove our main theorem. To obtain a construction time of our encoding, we first introduce a lemma which states a maximum number of nodes in $D_A^k$.

**Lemma 7.** (+) Given $2 \times n$ array $A$ and DAG $D_A^k$, there are at most $6kn$ nodes in $D_A^k$.

**Theorem 8.** (+) Given a $2 \times n$ array $A$, if there exists an $S(n,k)$-bit encoding to answer sorted 2-sided $\text{Top-k}$ queries on a 1D array of size $n$ in $T(n,k)$ time and such encoding can be constructed in $C(n,k)$ time, then we can encode $A$ in $2S(n,k) + 4n$ bits using $O(C(n,k) + k^2n^2 + knT(n,k))$ time.

We can obtain an encoding for answering sorted 4-sided $\text{Top-k}$ queries on an $m \times n$ array by extending the encoding of a $2 \times n$ array described in Theorem 8 as stated below.
Theorem 9. (*) Given an $m \times n$ array $A$, if there exists an $S(n,k)$-bit encoding to answer sorted 2-sided Top-$k$ queries on a 1D array of size $n$, then we can encode $A$ in $mS(n,k) + 2nm(m-1)$ bits, to support sorted 4-sided Top-$k$ queries on $A$.

Data structure for answering 4-sided Top-$k$ queries on $2 \times n$ array. The encoding of Theorem 8 shows that $4n$ bits are sufficient for answering Top-$k$ queries whose range spans both rows, when encodings for answering sorted 2-sided Top-$k$ queries for each row are given. However, this encoding does not support queries efficiently (takes $O(k^2n^2 + knT(n,k))$ time) since we need to reconstruct all the nodes in $D_A$ to answer a query (in the worst case). We now show that the query time can be improved to $O(k^2 + kT(n,k))$ time if we use $(4k+7)n + kn O(n \log kn)$ additional bits. Note that if we simply use the data structure of Grossi et al. [10] (which takes $44n \log k + O(n \log kn)$ bits to encode a 1D array of length $n$ to support Top-$k$ queries in $O(k)$ time) on the 1D array of size $2n$ obtained by writing the values of $A$ in column-major order, we can answer Top-$k$ queries on $A$ in $O(k)$ time using $88n \log k + O(n \log kn)$ additional bits. Although our data structure takes more query time and takes asymptotically more space, it uses less space for small values of $k$ (note that $4k + 7 < 88 \log kn$ for all integers $2 \leq k < 160$) when $n$ is sufficiently large. We now describe our data structure.

We first define a graph $G_{12} = \langle V(G_{12}), E(G_{12}) \rangle$ on $A$ as follows. The set of vertices $V(G_{12}) = \{1, 2, \ldots, n\}$, and there exists an edge $(i, j) \in E(G_{12})$ if and only if (i) $i < j$ and $A[1][i] < A[2][j]$, (ii) there are at most $k - 1$ positions in $A[1,2][i \ldots j]$ whose corresponding values are larger than both $A[1][i]$ and $A[2][j]$, and (iii) there is no vertex $j' > i$ that satisfies the condition (ii) such that $A[1][j] < A[2][j'] < A[2][j]$. We also define a graph $G_{21}$ on $A$ which is analogous to $G_{12}$. Each of the graphs $G_{12}$ and $G_{21}$ have $n$ vertices and at most $n$ edges. Also for any vertex $v \in V(G_{12})$ (resp., $V(G_{21})$), there exists at most one vertex $v'$ in $G_{12}$ (resp., $G_{21}$) such that $v$ is incident to $v'$ and $v < v'$. We now show that $G_{12}$ (thus, also $G_{21}$) is a $k$-page graph, i.e. there exist no $k + 1$ edges $(i_1, j_1) \ldots (i_{k+1}, j_{k+1}) \in E(G_{12})$ such that $i_1 < i_2 < \cdots < i_{k+1} < j_1 < j_2 < \cdots < j_{k+1}$. 

Lemma 10. Given $2 \times n$ array $A$, a graph $G_{12}$ on $A$ is a $k$-page graph.

Proof. Suppose that there are $k + 1$ edges $(i_1, j_1) \ldots (i_{k+1}, j_{k+1}) \in E(G_{12})$ such that $i_1 < i_2 < \cdots < i_{k+1} < j_1 < j_2 < \cdots < j_{k+1}$, and for $1 \leq t \leq k + 1$, let $i_t$ be a position of the minimum element in $A[1][i_t \ldots i_{k+1}]$. Then by the definition fo $G_{12}$, there are at least $k$ positions $(1, i_{t+1}), \ldots, (i_{k+1}), (2,j_1), \ldots, (2,j_{k+1})$ in $A[1,2][i_1 \ldots j_2]$ whose corresponding values in $A$ are larger than both $A[1][i_t]$ and $A[2][j_t]$, which contradicts the definition of $G_{12}$. △

From the above lemma and the succinct representation of $k$-page graphs of Munro and Raman [15] (with minor modification as described in [6]), we can encode $G_{12}$ and $G_{21}$ using $(4k+4)n + kn O(n \log kn)$ bits in total, and for any vertex $v \in V(G_{12}) \cup V(G_{21})$, we can find a vertex with the largest index which incident to $v$ in $O(k)$ time. Also to compare the elements in the same column, we maintain a bit string $P_A[1 \ldots n]$ of size $n$ such that for $1 \leq i \leq n$, $P_A[i] = 0$ if and only if $A[1][i] > A[2][i]$. Finally, for $G_{12}$ (resp., $G_{21}$), we maintain another bit string $Q_{12}[1 \ldots n-1]$ (resp., $Q_{21}[1 \ldots n-1]$) such that for $1 \leq i \leq n-1$, $Q_{21}[i] = 1$ (resp., $Q_{21}[i] = 1$) if and only if all elements in $A_2[i+1 \ldots n]$ (resp., $A_1[i+1 \ldots n]$) are smaller than $A[1][i]$ (resp., $A[2][i]$). We now show that if there is an encoding which can answer the sorted Top-$k$ queries on each row, then the encoding of $G_{12}$, $G_{21}$, and the additional arrays defined above are enough to answer 4-sided Top-$k$ queries on $A$. 


4 Lower bounds for encoding range Top-$k$ queries on $2 \times n$ array

In this section, we consider the lower bound on space for encoding a $2 \times n$ array $A$ to support Top-$k$ queries, when $k > 1$. Specifically for $1 \leq i \leq j \leq n$, we consider to lower bound on extra space for answering i) unsorted and sorted 3-sided Top-$k(1,2,1,i)$ queries, assuming that we have access to the encodings of the individual rows of $A$ that can answer unsorted or sorted 1-sided Top-$k$ queries and ii) sorted 4-sided Top-$k(1,2,i,j)$ queries, assuming that we have access to the encodings of the individual rows of $A$ that can answer sorted 2-sided Top-$k$ queries. We show that for answering unsorted (or sorted) 3-sided Top-$k(1,2,1,i)$ queries on $A$, at least $1.27n - o(n)$ (or $2n - O(\log n)$) extra bits are necessary, and for answering unsorted or sorted 4-sided Top-$k(1,2,i,j)$ queries on $A$, at least $2n - O(\log n)$ extra bits are necessary.

For simplicity (to avoid writing floors and ceilings, and to avoid considering some boundary cases), we assume that $k$ is even. (Also, if $k$ is odd we can consider the lower bound on extra space for answering 3-sided Top-$k$ queries as the lower bound of extra space for answering 3-sided Top-$(k-1)$ queries — it is clear that former one requires more space.) For both unsorted and sorted query cases, we assume that all elements in $A$ are distinct, and come from the set $\{1, 2, \ldots, 2n\}$; and also that each row in $A$ is sorted in the ascending order. Finally, for $1 \leq \ell \leq 2n$, we define the mapping $A^{-1}(\ell) = (i,j)$ if and only if $A[i][j] = \ell$.

Unsorted 3-sided Top-$k$ query. If $n \leq k/2$ we do not need any extra space since all positions are answers of unsorted Top-$k(1,2,1,i)$ queries for $i \leq n$. If not ($n > k/2$), for $1 \leq i \leq n - k/2$, let $U_i$ be a set of arrays of size $2 \times n$ such that i) for any $B \in U_i$, all of $\{1,2,\ldots,2i\}$ are in $B[1][1],B[1][2]\ldots,i$ and each row in $B$ is sorted in the ascending order, and ii) for any two distinct arrays $B,C \in U_i$, there exists $1 \leq j \leq i$ such that $B^{-1}(2j-1),B^{-1}(2j),C^{-1}(2j)-1) \neq \{C^{-1}(2j-1),C^{-1}(2j))$. By the definition of $U_i$, it is easy to show that for any two distinct arrays $B,C \in U_i$, unsorted Top-$k(1,2,1,k/2+j,B) \neq Top-k(1,2,1,k/2+j,C)$ if $\{B^{-1}(2j-1),B^{-1}(2j)\} \neq \{C^{-1}(2j-1),C^{-1}(2j)\}$ for some $j \leq i$. We compute the size of $U_i$ as follows. $|U_i| = 1$ since there exists only one case as $\{B^{-1}(1),B^{-1}(2)\} = \{(1,1),(2,1)\}$. For $i = 2$, we can consider three cases as $(1,2,3,4), (1,3,2,4), (1,4,2,3)$ if we write the elements of $B[1][1],B[2]$ in $U_2$ in row-major order (note that each row is sorted in ascending order). By computing the size of $U_i$ for $2 < i \leq n - k/2$, we obtain a following theorem.

Theorem 12. Given a $2 \times n$ array $A$ and encodings for answering unsorted (or sorted) 1-sided Top-$k$ queries on both rows in $A$, at least $[(n-k/2)] + o(n)$ additional bits are necessary for answering unsorted 3-sided Top-$k$ queries on $A$.

Proof. (Sketch) Since we need at least $\log |U_{n-k/2}|$ bits of extra space for answering unsorted Top-$k$ queries which span both rows, we only need to compute the size of $U_{n-k/2}$. To compute this, for $2 < i \leq n - k/2$, we construct the arrays in $U_i$ from the arrays in $U_{i-1}$, and obtain the recurrence relation: $|U_i| = 3|U_{i-2}| + 2|U_{i-1}| - |U_{i-2}|$. Solving this gives us the stated bound. Details of the proof are omitted due to space limitation.

Sorted 3-sided and 4-sided Top-$k$ query. In this case we divide a $2 \times n$ array $A$ into $2n/k$ blocks $A_1 \ldots A_{2n/k}$ of size $2 \times k/2$ as for $1 \leq \ell \leq k/2$, $A_\ell[i][j] = A[i][2(\ell - 1) + j]$ and all values of $A_\ell$ are in $\{k(\ell - 1) + 1 \ldots k\}$. Then for any $2 \times n$ array $A$ and
Encoding Two-Dimensional Range Top-$k$ Queries Revisited

$A'$, sorted Top-$k(1, 2, k(i - 1)/2 + 1, ki/2, A) \neq$ Top-$k(1, 2, k(i - 1)/2 + 1, ki/2, A')$, and
Top-$k(1, 2, 1, ki/2, A) \neq$ Top-$k(1, 2, 1, ki/2, A')$ if $A_i \neq A'_i$ for $1 \leq i \leq 2n/k$. Let $S_i$ be the set
of arrays of size $2 \times i$ such that for any $B \in S_i$, all values of $B$ are in $\{1, 2i\}$ and both rows of
$B$ are sorted in ascending order. Since the size of $S_i$ is same as central binomial number, $\binom{2i}{i}$,
which is well-known as at least $4^i/\sqrt{i}$ [13]. Therefore, at least $[2n \log |S_{k/2}|/k] \geq 2n - O(\log n)$
bits are necessary for answering sorted Top-$k$ queries that span both the rows, when encodings
for answering sorted (or unsorted) on both rows are given.

**Theorem 13.** Given a $2 \times n$ array $A$, at least $2n - O(\log n)$ additional bits are necessary
for answering sorted 3-sided (resp., 4-sided) Top-$k$ queries on $A$ if encodings for answering
unsorted (or sorted) 1-sided (resp., 2-sided) Top-$k$ queries on both rows in $A$ are given.

**5 Conclusion**

In this paper, we proposed encodings for answering Top-$k$ queries on 2D arrays. For $2 \times n$
arrays, we proposed upper and lower bounds on space for answering 3-sided sorted and
unsorted Top-$k$ queries. Finally, we obtained an $(m \log \left(\frac{k+1}{n}\right) + 2mn(m-1) + o(n))$-bit encoding for answering 4-sided sorted Top-$k$ queries on $m \times n$ arrays. We end with
the following open problems: (a) can we support 4-sided sorted Top-$k$ queries with efficient query
time on $m \times n$ arrays using less than $O(mn \log n)$ bits when $m = o(\log n)$? (b) is there any
improved lower or upper bound for answering 4-sided sorted Top-$k$ queries on $2 \times n$ arrays?

---

**References**

1. Omer Berkman and Uzi Vishkin. Recursive Star-Tree Parallel Data Structure. *SIAM J.

2. Gerth Stølting Brodal, Andrej Brodnik, and Pooya Davoodi. The Encoding Complexity
   Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings*,

   doi:10.1007/s00453-011-9499-0.

   minima and range top-2 queries. *Philosophical Transactions of the Royal Society of London
   2013.0131.

5. Johannes Fischer and Volker Heun. Finding Range Minima in the Middle: Approximations
   s11786-009-0007-8.

   Queries and Applications. In *Combinatorial Pattern Matching - 26th Annual Symposium,
   doi:10.1007/978-3-319-19929-0_17.

   Selection, and Min-Max. In *Automata, Languages, and Programming - 42nd International
   Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, pages 593–

8. Mordecai J. Golin, John Iacono, Danny Krizanc, Rajeev Raman, Srinivasa Rao Satti, and


Abstract

A border $u$ of a word $w$ is a proper factor of $w$ occurring both as a prefix and as a suffix. The maximal unbordered factor of $w$ is the longest factor of $w$ which does not have a border. Here an $O(n \log n)$-time with high probability (or $O(n \log n \log \log n)$-time deterministic) algorithm to compute the Longest Unbordered Factor Array of $w$ for general alphabets is presented, where $n$ is the length of $w$. This array specifies the length of the maximal unbordered factor starting at each position of $w$. This is a major improvement on the running time of the currently best worst-case algorithm working in $O(n^{1.5})$ time for integer alphabets [Gawrychowski et al., 2015].

1 Introduction

There are two central properties characterising repetitions in a word --period and border-- which play direct or indirect roles in several diverse applications ranging over pattern matching, text compression, assembly of genomic sequences and so on (see [3, 6]). A period of a non-empty word $w$ of length $n$ is an integer $p$ such that $1 \leq p \leq n$, if $w[i] = w[i+p]$, for all $1 \leq i \leq n - p$. For instance, 3, 6, 7, and 8 are periods of the word aabaabaa. On the other hand, a border $u$ of $w$ is a (possibly empty) proper factor of $w$ occurring both as a prefix and as a suffix of $w$. For example, $\varepsilon$, a, aa, and aabaa are the borders of $w = aabaabaa$.

In fact, the notions of border and period are dual: the length of each border of $w$ is equal to the length of $w$ minus the length of some period of $w$. For example, aa is a border of the word aabaabaa; it corresponds to period 6 = |aabaabaa| − |aa|. Consequently, the basic data
structure of periodicity on words is the border array which stores the length of the longest border for each prefix of \( w \). The computation of the border array of \( w \) was the fundamental concept behind the first linear-time pattern matching algorithm – given a word \( w \) (pattern), find all its occurrences in a longer word \( y \) (text). The border array of \( w \) is better known as the “failure function” introduced by Knuth, Morris, and Pratt [12]. It is well-known that the border array of \( w \) can be computed in \( O(n) \) time, where \( n \) is the length of \( w \), by a variant of the Knuth-Morris-Pratt algorithm [12].

Another notable aspect of the inter-dependency of these dual notions is the relationship between the length of the maximal unbordered factor of \( w \) and the periodicity of \( w \). A maximal unbordered factor is the longest factor of \( w \) which does not have a non-empty border; its length is usually represented by \( \mu(w) \), e.g. the maximal unbordered factor is \( \text{aabab} \) and \( \mu(w) = 5 \) for the word \( w = \text{baabab} \). This dependency has been a subject of interest in the literature for a long time, starting from the 1979 paper of Ehrenfeucht and Silberger [9] in which they raised the question – at what length of \( w \), \( \mu(w) \) is maximal (i.e., equal to the minimal period of the word as it is well-known that it cannot be longer than that). This line of questioning, after being explored for more than three decades, culminated in 2012 with the work by Holub and Nowotka [11] where an asymptotically optimal upper bound \( (\mu(w) \leq \frac{3}{2}n) \) was presented; the historic overview of the related research can be found in [11].

Somewhat surprisingly, the symmetric computational problem – given a word \( w \), compute the longest factor of \( w \) that does not have a border – had not been studied until very recently. In 2015, Kucherov et al. [15] considered this arguably natural problem and presented the first sub-quadratic-time solution. A naïve way to solve this problem is to compute the border array starting at each position of \( w \) and locating the rightmost zero, which results in an algorithm with \( O(n^2) \) worst-case running time. On the other hand, the computation of the maximal unbordered factor can be done in linear time for the cases when \( \mu(w) \) or its minimal period is small (i.e., at most half the length of \( w \) using the linear-time computation of unbordered conjugates [8]). However, as has been illustrated in [15] and [2], most of the words do not fall in this category owing to the fact that they have large \( \mu(w) \) and consequently large minimal period. In [15], an adaptation of the basic algorithm has been provided with average-case running time \( O(n^2/\sigma^4) \), where \( \sigma \) is the alphabet’s size; it has also been shown to work better, both in practice and asymptotically, than another straightforward approach that employs data structures from [14, 13] to query all relevant factors.

The currently fastest worst-case algorithm to compute the maximal unbordered factor of a given word takes \( O(n^{1.5}) \) time; it was presented by Gawrychowski et al. [10] and it works for integer alphabets (alphabets of polynomial size in \( n \)). This algorithm works by categorising bordered factors into short borders and long borders depending on a threshold, and exploiting the fact that, for each position, the short borders are bounded by the threshold and the long borders are small in number. The resulting algorithm runs in \( O(n \log n) \) time on average. More recently, an \( O(n) \)-time average-case algorithm was presented using a refined bound on the expected length of the maximal unbordered factor [2].

Our Contribution. In this paper, we show how to efficiently answer the Longest Unbordered Factor question using combinatorial insight. Specifically, we present an algorithm that computes the Longest Unbordered Factor Array in \( O(n \log n) \) time with high probability. The algorithm can also be implemented deterministically in \( O(n \log n \log^2 \log n) \) time. This array specifies the length of the maximal unbordered factor at each position in \( w \). We thus improve on the running time of the currently fastest algorithm, which reports only the maximal unbordered factor of \( w \) and works only for integer alphabets, taking \( O(n^{1.5}) \) time.
Structure of the Paper. In Section 2, we present the preliminaries, some useful properties of unbordered words, the algorithmic toolbox, and a formal definition of the problem. We lay down the combinatorial foundation of the algorithm in Section 3 and expound the algorithm in Section 4; its analysis is explicated in Section 5. We conclude this paper with a final remark in Section 6.

2 Background

Definitions and Notation. We consider a finite alphabet $\Sigma$ of letters. Let $\Sigma^*$ be the set of all finite words over $\Sigma$. The empty word is denoted by $\varepsilon$. The length of a word $w$ is denoted by $|w|$. For a word $w = w[1]w[2] \ldots w[n]$, $w[i \ldots j]$ denotes the factor $w[i]w[i+1] \ldots w[j]$, where $1 \leq i \leq j \leq n$. The concatenation of two words $u$ and $v$ is the word composed of the letters of $u$ followed by the letters of $v$. It is denoted by $uv$ or also by $u \cdot v$ to show the decomposition of the resulting word. Suppose $w = uv$, then $u$ is a prefix and $v$ is a suffix of $w$; if $u \neq w$ then $u$ is a proper prefix of $w$; similarly, if $v \neq w$ then $v$ is a proper suffix of $w$. Throughout the paper we consider a non-empty word $w$ of length $n$ over a general alphabet $\Sigma$; in this case, we replace each letter by its rank such that the resulting word consists of integers in the range $\{1, \ldots, n\}$. This can be done in $O(n \log n)$ time after sorting the letters of $\Sigma$.

An integer $1 \leq p \leq n$ is a period of $w$ if and only if $w[i] = w[i+p]$ for all $1 \leq i \leq n - p$. The smallest period of $w$ is called the minimum period (or the period) of $w$, denoted by $\lambda(w)$. A word $u$ ($u \neq w$) is a border of $w$, if $w = uv = v'u$ for some non-empty words $v$ and $v'$; note that $u$ is both a proper prefix and a suffix of $w$. It should be clear that if $w$ has a border of length $|w| - p$ then it has a period $p$. Thus, the minimum period of $w$ corresponds to the length of the longest border (or the border) of $w$. Observe that the empty word $\varepsilon$ is a border of any word $w$. If $u$ is the shortest border then $u$ is the shortest non-empty border of $w$.

The word $w$ is called bordered if it has a non-empty border, otherwise it is unbordered. Equivalently, the minimum period $p = |w|$ for an unbordered word $w$. Note that every bordered word $w$ has a shortest border $u$ such that $w = uvu$, where $u$ is unbordered. By $\mu(w)$ we denote the maximum length among all the unbordered factors of $w$.

Useful Properties of Unbordered Words. Recall that a word $u$ is a border of a word $w$ if and only if $u$ is both a proper prefix and a suffix of $w$. A border of a border of $w$ is also a border of $w$. A word $w$ is unbordered if and only if it has no non-empty border; equivalently $\varepsilon$ is the only border of $w$. The following properties related to unbordered words form the basis of our algorithm and were presented and proved in [7].

Proposition 1 ([7]). Let $w$ be a bordered word and $u$ be the shortest non-empty border of $w$. The following propositions hold:
1. $u$ is an unbordered word;
2. $u$ is the unique unbordered prefix and suffix of $w$;
3. $w$ has the form $w = uvu$.

Proposition 2 ([7]). For any word $w$, there exists a unique sequence $(u_1, \ldots, u_k)$ of unbordered prefixes of $w$ such that $w = u_k \cdots u_1$. Furthermore, the following properties hold:
1. $u_1$ is the shortest border of $w$;
2. $u_k$ is the longest unbordered prefix of $w$;
3. for all $i$, $1 \leq i \leq k$, $u_i$ is an unbordered prefix of $u_k$. 

ISAAC 2018
The computation of the unique sequence described in Proposition 2 provides a unique unbordered-decomposition of a word. For instance, for \( w = \text{baabababab} \) the unique unbordered-decomposition of \( w \) is \( \text{baa} \cdot \text{ba} \cdot \text{b} \cdot \text{ba} \cdot \text{ba} \cdot \text{b} \).

**Longest Successor Factor (Length and Reference) Arrays.** Here, we present the arrays that will act as a toolbox for our algorithm. The longest successor factor of \( w \) (denoted by \( \text{LSF} \)) starting at position \( i \), is the longest factor of \( w \) that occurs at \( i \) and has at least one other occurrence in the suffix \( w[i+1..n] \). The longest successor factor array gives for each position \( i \) in \( w \), the length of the longest factor starting both at position \( i \) and at another position \( j > i \). Formally, the longest successor factor array (\( \text{LSF} \)) is defined as follows.

\[
\text{LSF}_i[i] = \begin{cases} 
0 & \text{if } i = n, \\
\max\{k \mid w[i..i+k-1] = w[j..j+k-1]\}, & \text{for } i < j \leq n.
\end{cases}
\]

Additionally, we define the \( \text{LSF-Reference Array} \), denoted by \( \text{LSF}_r \). This array specifies, for each position \( i \) of \( w \), the reference of the longest successor factor at \( i \). The reference of \( i \) is defined as the position \( j \) of the last occurrence of \( w[i..i+\text{LSF}_r[i]-1] \) in \( w \); we say \( i \) refers to \( j \). Formally, \( \text{LSF-Reference Array} \) (\( \text{LSF}_r \)) is defined as follows.

\[
\text{LSF}_r[i] = \begin{cases} 
\text{nil} & \text{if } \text{LSF}_r[i] = 0, \\
\max\{j \mid w[j..j+\text{LSF}_r[i]-1] = w[i..i+\text{LSF}_r[i]-1]\} & \text{for } i < j \leq n.
\end{cases}
\]

**Computation:** Note that the longest successor factor array is a mirror image of the well-studied longest previous factor array which can be computed in \( O(n) \) time for integer alphabets \([4, 5]\). Moreover, in \([4]\), an additional array that keeps a position of some previous occurrence of the longest previous factor was presented; such position may not be the leftmost. Arrays \( \text{LSF}_l \) and \( \text{LSF}_r \) can be computed using simple modifications (pertaining to the symmetry between the longest previous and successor factors) of this algorithm\(^1\) within \( O(n) \) time for integer alphabets.

**Example 3.** Let \( w = \text{aabbabaabbaabababab} \). The associated arrays are as follows.

\[
\begin{array}{cccccccccccccccc}
\text{i} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 \\
w[i] & a & a & b & b & a & b & a & a & b & b & a & a & b & a & b & b & a & b & a & b \\
\text{LSF}_l[i] & 5 & 6 & 5 & 4 & 3 & 4 & 3 & 4 & 3 & 2 & 1 & 4 & 3 & 2 & 1 & 3 & 2 & 1 & 0 & 0 \\
\text{LSF}_r[i] & 7 & 14 & 15 & 16 & 17 & 10 & 11 & 14 & 15 & 18 & 19 & 17 & 18 & 19 & 20 & 18 & 19 & 20 & \text{nil} & \text{nil} \\
\end{array}
\]

**Remark.** For brevity, we will use \( \text{LSF} \) and \( \text{LUF} \) to represent the longest successor factor and the longest unbordered factor, respectively.

**Problem Definition.** The **Longest Unbordered Factor Array** problem can be defined formally as follows.

\begin{center}
**LONGEST UNBORDERED FACTOR ARRAY**
\end{center}

**Input:** A word \( w \) of length \( n \).

**Output:** An array \( \text{LUF}[1..n] \) such that \( \text{LUF}[i] \) is the length of the maximal unbordered factor starting at position \( i \) in \( w \), for all \( 1 \leq i \leq n \).

---

\(^1\) The modified algorithm also computes some starting position \( j > i \) for each factor \( w[i..i+|\text{LSF}_r[i]|-1] \), \( 1 \leq i \leq n \). Each such factor corresponds to the lowest common ancestor of the two terminal nodes in the suffix tree of \( w \) representing the suffixes \( w[i..n] \) and \( w[j..n] \); this ancestor can be located in constant time after linear-time preprocessing \([1]\). A linear-time preprocessing of the suffix tree also allows for constant-time computation of the rightmost starting position of each such factor.
3 Combinatorial Tools

The core of our algorithm exploits the unique unbordered-decomposition of all suffixes of \( w \) in order to compute the length of the maximal (longest) unbordered prefix of each such suffix. Let the unbordered-decomposition of \( w[i...n] \) be \( u_k \cdots u_1 \) as in Proposition 2. Then \( LUF[i] = |u_k| \). In order to compute the unbordered-decomposition for all the suffixes efficiently, the algorithm uses the repetitive structure of \( w \) characterised by the longest successor factor arrays.

**Basis of the algorithm.** Abstractly, it is easy to observe that for a given position, if the length of the longest successor factor is zero (no factor starting at this position repeats afterwards) then the suffix starting at that position is necessarily unbordered. On the other hand, if the length of the longest successor factor is smaller than the length of the unbordered factor at the reference (the position of the the last occurrence of the longest successor factor) then the ending positions of the longest unbordered factors at this position and that at its reference will coincide; these two cases are formalised in Lemmas 5 and 6 below. The remaining case is not straightforward and its handling accounts for the bulk of the algorithm.

** Lemma 5.** If \( LSF_r[i] = 0 \) then \( LUF[i] = n - i + 1 \), for \( 1 \leq i \leq n \).

** Lemma 6.** If \( LSF_r[i] = j \) and \( LSF_r[i] < LUF[j] \) then \( LUF[i] = j + LUF[j] - i \), for \( 1 \leq i \leq n \).

*Proof.* Let \( k = j + LUF[j] - 1 \). We first show that \( w[i...k] \) is unbordered. Assume that \( w[i...k] \) is bordered and let \( \beta \) be the length of one of its borders (\( \beta < LSF_r[i] \) as \( LSF_r[i] = j \)). This implies that \( w[i...i+\beta-1] = w[k-\beta+1...k] \). Since \( w[i...i+LSF_r[i]-1] = w[j...j+LSF_r[i]-1] \), we get \( w[j...j+\beta-1] = w[k-\beta+1...k] \) (i.e., \( w[j...k] \) is bordered) which is a contradiction. Moreover, \( w[k+1...n] \) can be factorised into prefixes of \( w[j...k] \) (by definition of \( LUF \)); every such prefix is also a proper prefix of \( w[i...i+LSF_r[i]-1] \) which will make every factor \( w[i...k'], k < k' \leq n \), to be bordered. This completes the proof.

We introduce the notion of a hook to handle finding the unbordered-decomposition of suffixes \( w[i...n] \) for the remaining case (i.e., when \( LSF_r[i] \geq LUF[LSF_r[i]] \)).

** Definition 7 (Hook).** Consider a position \( j \) in a length-\( n \) word \( w \). Its hook \( H_j \) is the smallest position \( q \) such that \( w[q...j-1] \) can be decomposed into unbordered prefixes of \( w[j...n] \).

The following observation provides a greedy construction of this decomposition.

** Observation 8.** The decomposition of a word \( v \) into unbordered prefixes of another word \( u \) is unique. This decomposition can be constructed by iteratively trimming the shortest prefix of \( u \) which occurs as a suffix of the decomposed word.

Moreover, the decomposability into unbordered prefixes of \( u \) is hereditary in a certain sense:
Figure 1 Case $a$ ($i < q$): The unbordered-decomposition of $w[i \ldots n]$ consists of $w[i \ldots q - 1]$ as the longest unbordered prefix, followed by a sequence of unbordered prefixes of $u$, including $u$ itself at position $j$. Therefore, $\text{LUF}[i] = q - i$.

Observation 9. If a word $v$ can be decomposed into unbordered prefixes of $u$, then every prefix of $v$ also admits such a decomposition. Formally, if $v = u_r \cdot u_{r-1} \cdot \ldots \cdot u_2 \cdot u_1$ such that each $u_i, r \geq i \geq 1$, is an unbordered prefix of $u$ then any prefix $v[1 \ldots k]$ can be uniquely decomposed as $v[1 \ldots k] = u_r \cdot u_{r-1} \cdot \ldots \cdot u_i \cdot u_i' \cdot u_i'' \cdot \ldots \cdot u_1'$, where $k$ falls in $u_i$ and each $u_i', p \geq i \geq 1$, is an unbordered prefix of $u$; simply, the decomposition preceding $u_i$ will be retained by the prefix.

Example 10. Consider $w = \text{aabbabaabbaababab}$ as in Example 4. Observe that $H_{18} = 13$: the factor $w[13 \ldots 17] = ba \cdot b \cdot ba$ can be decomposed into unbordered prefixes of $w[18 \ldots 20] = bab$. Moreover, no prefix of $w[18 \ldots 20]$ matches a suffix of $w[1 \ldots 12] = \cdots aa$.

The hook $H_j$ has its utility when $j$ is a reference as shown in the following lemma.

Lemma 11. Consider a position $i$ such that $\text{LSF}_\ell[i] \geq \text{LUF}[j]$, where $j = \text{LSF}_r[i]$. Then

\[
\text{LUF}[i] = \begin{cases} 
H_j - i & \text{if } i < H_j, \\
\text{LUF}[j] & \text{otherwise.}
\end{cases}
\]

Proof. Let $u = w[i \ldots j] + \text{LUF}[j] - 1$, $v = w[i \ldots i + \text{LSF}_\ell[i] - 1]$, and $q = H_j$. Observe that $u$ occurs at position $i$ and that $w[q \ldots n]$ can be decomposed into unbordered prefixes of $u$.

Case $a$: $i < q$. We shall prove that $w[i \ldots q - 1]$ is the longest unbordered prefix of $w[i \ldots n]$; see Figure 1. By Observation 9, any longer factor $w[i \ldots k]$, $q \leq k \leq n$ has a suffix $w[q \ldots k]$ composed of unbordered prefixes of $u$. Thus, $w[i \ldots k]$ must be bordered, because $u$ is its prefix. To conclude, for a proof by contradiction suppose that $w[i \ldots q - 1]$ has a border $v'$. Note that $|v'| \leq \text{LSF}_\ell[i]$, so $v'$ is a prefix of $v$. Hence, it occurs both as a suffix of $w[1 \ldots q - 1]$ and a prefix of $w[j \ldots n]$, which contradicts the greedy construction of $q = H_j$ (Observation 8).

Case $b$: $i \geq q$. The decomposition of $w[q \ldots n]$ into unbordered prefixes of $u$ yields a decomposition of $w[i \ldots n]$ into unbordered prefixes of $u$, starting with $u$. This is the unbordered-decomposition of $w[i \ldots n]$ (see Proposition 2), which yields $\text{LUF}[i] = |u| = \text{LUF}[j]$.

4 Algorithm

The algorithm operates in two phases: a preprocessing phase followed by the main computation phase. The preprocessing phase accomplishes the following: Firstly, compute the longest successor factor array $\text{LSF}_\ell$ together with $\text{LSF}_r$ array. If $\text{LSF}_r[i] = j$ then we say $i$ refers to $j$ and mark $j$ in a boolean array (LsReference) as a reference.

In the main phase, the algorithm computes the lengths of the longest unbordered factors for all positions in $w$. Moreover, it determines $\text{HOOK}[j] = H_j$ for each potential reference, i.e., each position $j$ such that $j = \text{LSF}_r[i]$ and $\text{LSF}_\ell[i] \geq \text{LUF}[j]$ for some $i < j$; see Lemma 11.
we iteratively compute and truncate the shortest prefixes of \( \ell \) exceeding \( w \) of time a prefix of length greater than or equal to \( u \) leftwards (towards its final value). Consider the starting position done for \( \text{FindHook} \).

Moreover, the same termination is and terminating as soon as no prefix of the considered factor; shortening the length of the considered factor in each iteration is a potential reference, then \( \text{OOK}[i] \) is also computed, as described in Section 4.1. It is evident that the computational phase of the algorithm fundamentally reduces to finding the hooks for potential references; for brevity, the term reference will mean a potential reference hereafter.

4.1 Finding Hook (FindHook Function)

Main idea. When \( \text{FindHook} \) is called on a reference \( j \), it must return \( H_j \). A simple greedy approach follows directly from Observation 8; see also Figure 2. Initially, the factor \( w[1 \ldots j - 1] \) is considered and the shortest suffix of \( w[1 \ldots j - 1] \) which is a prefix of \( w[j \ldots n] \) is computed. Then this suffix, denoted \( u_1 = w[i_1 \ldots j - 1] \), is truncated (chopped) from the considered factor \( w[1 \ldots j - 1] \); the next factor considered will be \( w[1 \ldots i_1 - 1] \). In general, we iteratively compute and truncate the shortest prefixes of \( w[j \ldots n] \) from the right end of the considered factor; shortening the length of the considered factor in each iteration and terminating as soon as no prefix of \( w[j \ldots n] \) can be found. If the considered factor at termination is \( w[1 \ldots q - 1] \), position \( q \) is returned by the function as \( H_j \).

The factors \( w[q \ldots j - 1] \) considered by successive calls of \( \text{FindHook} \) function may overlap. Moreover, the same chains of consecutive unbordered prefixes may be computed several times throughout the algorithm. To expedite the chain computation in the subsequent calls of \( \text{FindHook} \) on another reference \( j' \) (\( j' < j \)), we can recycle some of the computations done for \( j \) by shifting the value \( \text{OOK}[] \) of each such index (at which a prefix was cut for \( j \)) leftwards (towards its final value). Consider the starting position \( i_k \) at which \( u_k = w[i_k \ldots i_k - 1] \) is the shortest unbordered prefix of \( w[j \ldots n] \) computed at \( i_k - 1 \). Let \( i_p \) be the first position considered after \( i_k \) such that \( |u_p| > |u_k| \). In this case, every factor \( u_{k+1}, \ldots, u_{p-1} \) is a prefix of \( u_k \); see Figure 2. Therefore, \( w[i_{p-1} \ldots i_k - 1] \) can be decomposed into prefixes of \( u_k \) (and of \( w[i_k \ldots n] \)). Consequently, we set \( \text{OOK}[i_k] = i_{p-1} \) so that the next time a prefix of length greater than or equal to \( |u_k| \) is cut at \( i_k \), we do not have to repeat truncating the prefixes \( u_{k+1}, \ldots, u_{p-1} \) and we may start directly from position \( i_{p-1} \).

In order to express the intermediate values in the \( \text{OOK} \) table, we generalize the notion of \( H_j \): for a position \( j \) and a length \( \ell \), we define \( H^\ell_j \) as the smallest position \( q \) such that \( w[q \ldots j - 1] \) can be decomposed into unbordered prefixes of \( w[j \ldots n] \) whose lengths do not exceed \( \ell \). Observe that \( H^0_j = j \) and \( H^\ell_j = H_j \) if \( \ell \geq \text{LUF}[j] \).
Implementation. For each position $i_k$, we set $\text{HOOK}[i_k] = H_{|i_k|}$, equal to $i_{p-1}$ in the case considered above. Computing these values for all indices $i_k$ can be efficiently realised using a stack. Every starting position $i_p$, at which $u_p$ is cut, is pushed onto the stack as a (length, position) pair $(|u_p|, i_p)$. Before pushing, every element $(|u_k|, i_k)$ such that $|u_k| < |u_p|$ is popped and the hook value of index $i_k$ is updated ($\text{HOOK}[i_k] = H_{|u_k|} = i_{p-1} = i_p + |u_p|$).

Analysis. Throughout the algorithm, each unbordered prefix $u_p$ at position $i_p$ is computed just once by the $\text{FindHook}$ function. Nevertheless, a longer unbordered prefix $u'_p$ may be computed at $i_p$ again when $\text{FindHook}$ is called on reference $j'$ (where $q < j' < j$).

In what follows, we introduce certain characteristics of the computed unbordered prefixes which aids in establishing the relationship between the stacks of various references. Let $S_j$ be the set of positions pushed onto the stack during a call of $\text{FindHook}$ on reference $j$.

Definition 12 (Twin Set). A twin set of reference $j$ for length $\ell$, denoted by $T_j^{\ell}$, is the set of all the positions $i \in S_j$ which were pushed onto the stack paired with length $\ell$ in the call of $\text{FindHook}$ on reference $j$ (i.e., $T_j^{\ell} = \{i \mid (\ell, i) \text{ was pushed onto the stack of } j\}$).

Note that a unique shortest unbordered prefix of $w[j..LUF[j] - 1]$ occurs at each $i$ belonging to the same twin set. However, as and when a longer prefix at $i$ is cut (say $\ell'$) for another reference $j' < j$, $i$ will be added to $T_j^{\ell'}$.

Remark. $S_j = \bigcup_{\ell=1}^{\text{LUF}[j]} T_j^{\ell}$.

Hereafter, a twin set will essentially imply a non-empty twin set.

Lemma 13. If $j'$ and $j$ are references such that $j' \in S_j$, then $\mathcal{H}_j \leq \mathcal{H}_{j'}$.

Proof. Since $j' \in S_j$, the suffix $w[j'..n]$ (and, by Observation 9, its every prefix $w[j'..k]$) can be decomposed into unbordered prefixes of $w[j..n]$. Consequently, any decomposition into unbordered prefixes of $w[j'..n]$ yields a decomposition into unbordered prefixes of $w[j..n]$. In particular, $w[\mathcal{H}_{j}..n]$ admits such a decomposition, which implies $\mathcal{H}_j \leq \mathcal{H}_{j'}$.

If the stack $S_j$ is the most recent stack containing a reference $j'$, we say that $j'$ is the parent of $j$. More formally, the parent of $j'$ is defined as $\min\{j \mid j' \in S_j\}$. If $j'$ does not belong to any stack (and thus has no parent), we will call it a base reference.

Lemma 14. If $j$ and $j'$ are two references such that $j$ is the parent of $j'$ and $j' \in T_j^{\ell}$, then each position $i \in S_{j'}$ satisfies the following properties:

1. $i \in T_j^{\ell}$;
2. there exists $k \in T_j^{\ell'}$, with $\ell' > \ell$, such that $(k + \ell' - i, i)$ is pushed onto the stack of $j'$.

Proof. Let $p$ be the value of $\text{HOOK}[j']$ prior to the execution of $\text{FindHook}(j')$. Since $j' \in T_j^{\ell}$, the earlier call $\text{FindHook}(j)$ has set $\text{HOOK}[j'] = \mathcal{H}_{j'}$. As $j'$ is the parent of $j'$, no further call has updated $\text{HOOK}[j']$. Thus, we conclude that $p = \mathcal{H}_{j'}$.

Consequently, the first pair pushed onto the stack of $j'$ is $(|z|, i)$, where $z = w[i..p-1]$ is the shortest suffix of $w[1..p-1]$ which also occurs as a prefix of $w[j'..n]$ (see Figure 3). Moreover, observe that $|z| > \ell$ by the greedy construction of $\mathcal{H}_{j'}$.

\[^2\text{It will be easy to deduce after Lemma 14 that the length of the prefix cut (the next time) at the same position will be at least twice the length of the current prefix cut at it.}\]
The proof is trivial if \( j' \in \mathcal{T}_j^T \). 

\textbf{Lemma 15.} If \( j \) is the parent of two references \( j'' < j' \), both of which belong to \( \mathcal{T}_j^T \), then \( \mathcal{S}_j \cap \mathcal{S}_{j''} = \emptyset \).

\textbf{Proof.} The proof is trivial if \( \ell = \text{LUF}[j] \). Let \( \ell < \text{LUF}[j] \), \( u = w[j..j+\text{LUF}[j]-1] \) and \( v \) be the shortest unbordered prefix of \( u \) cut at \( j' \) and \( j'' \) (i.e., \( |v| = \ell \)). Let \( w' = w[j'..j'+\text{LUF}[j']-1] \) and \( w'' = w[j'..'j''+\text{LUF}[j'']-1] \). Here, the current call to the \textsc{FindHook} function has been made on the reference \( j'' \). Consider the largest position \( i \) such that it is common to the stacks of \( j' \) and \( j'' \) i.e. \( i \in \mathcal{S}_j \) and \( i \in \mathcal{S}_{j''} \). Let the prefixes cut at \( i \) be \( z_1 = w[i..p] \) and \( z_2 = w[i..k] \). Observe that \( i \) being the largest position and \( j' \neq j'' \) ensure that \( |z_1| \neq |z_2| \). Without loss of generality, let \( |z_1| < |z_2| \) (examine Figure 4).

1. \( j' \) cuts \( z_2 \) and \( j'' \) cuts \( z_1 \): We proceed with the proof below by showing that there is a reference between \( j' \) and \( j \) that pushes \( j' \) onto its stack, thus contradicting the fact that \( j \) is the parent of \( j' \).
Following Observation 9, \( w[i..k] \) can be decomposed into unbordered prefixes of \( u'' \) with the first prefix being \( z_1 \) i.e. \( z_2 = z_1 \cdot x_1 \cdot x_2 \ldots \cdot x_r \). Here, \( |x_r| > |z_1| \) otherwise \( z_2 \) is bordered. Moreover, each \( x_r \) larger than \( v \) has corresponding position in \( S_r \) and others (i.e. \( |z_2| \leq |v| \)) are skipped because of HOOK[]. Let \( x_s \) be the first of these \( x_i, 1 \leq i \leq r \) such that \( |x_s| > |z_1| \); the prefix \( z_2 = z_1 \ldots x_s \) is unbordered. In the occurrence of \( z_2 \) at \( j' \), let \( j_0 \) be the position corresponding to \( x_s \) i.e. \( j_0 = j' + |z_1| \ldots x_{s-1} | \).

Note that \( x_s \), like every \( x_i \) and \( z_1 \), has \( v \) as proper prefix and some \( v_i \) as a proper suffix where \( v_i \) is an unbordered prefix of \( u \) longer than \( v \) (from Lemma 14). Therefore, \( j_0 < j \) (\( x_s \) cannot start at \( j \) otherwise it would be bordered and \( x_s \) starting after \( j \) would contradict the assumption that \( j \) is the parent of \( j' \) as \( w[j'..j_0] \) can be factorised into prefixes of \( x_s \)).

Now, we prove that \( j_0 \) is a (potential) reference. The fact that \( j' \) is a potential reference ensures that \( \tilde{u} = w[j_0..j' + |u'| - 1] \) is a repeated factor. Moreover, \( \tilde{u} \) contains the luf at \( j_0 \), say \( u_0 \), because \( u_0 \) is a factor (or suffix) of \( u' \) (since \( w[j'..j_0 - 1] \) can be decomposed into prefixes of \( x_s \)); an implication is that \( |\tilde{u}| \geq |u_0| \). Thus, \( j_0 \) is a reference if the last occurrence of \( \tilde{u} \) is at \( j_0 \). For contradiction, assume that the factor \( \tilde{u} \) has another occurrence at some position larger than \( j_0 \). This implies that there is another occurrence of \( u \) as \( u_0 \) contains \( u \) (the luf at any position which is in the stack of \( j \), ends at or after \( j + |u'| - 1 \)). It is not possible as the last of the occurrences of \( u \) after \( j \) would cause \( j, j' \), \( j'' \) etc. to go in its stack and \( j \) would no longer be the parent of \( j' \) or \( j'' \).

Summing up, \( j_0 < j \) is a reference with \( x_s \) as a prefix of \( u_0 \). If \( j \) is the parent of \( j_0 \) then \( j_0 \) would have pushed \( j' \) onto its stack, otherwise another reference \( j_1 < j \), \( j_0 < j_1 < j \) that pushed \( j_0 \) onto its stack would have pushed \( j' \) as well. In either case, \( j \) is not the parent of \( j' \) which is a contradiction.

2. \( j' \) cuts \( z_1 \) and \( j'' \) cuts \( z_2 \): Using the similar argument as in Case 1, we can prove that this case would lead to the conclusion that there is another reference between \( j'' \) and \( j \) that would push \( j'' \) onto its stack and hence contradicting that \( j \) is the parent of \( j'' \).

### 4.2 Finding Shortest Border (FindBeta Function)

Given a reference \( j \) and a position \( q \), function \textsc{FindBeta} returns the length \( \beta \) of the shortest prefix of \( w[j..n] \) that is a suffix of \( w[1..q - 1] \), or \( \beta = 0 \) if there is no such prefix; note that the sought shortest prefix is necessarily unbordered.

To find this length, we use ‘prefix-suffix queries’ of [14, 13]. Such a query, given a positive integer \( d \) and two factors \( x \) and \( y \) of \( w \), reports all prefixes of \( x \) of length between \( d \) and \( 2d \) that occur as suffixes of \( y \). The lengths of sought prefixes are represented as an arithmetic progression, which makes it trivial to extract the smallest one. A single prefix-suffix query can be implemented in \( \mathcal{O}(1) \) time after randomized preprocessing of \( w \) which takes \( \mathcal{O}(n) \) time in expectation [14], or \( \mathcal{O}(n \log n) \) time with high probability [13]. Additionally, replacing hash tables with deterministic dictionaries [16], yields an \( \mathcal{O}(n \log n \log^2 \log n) \)-time deterministic preprocessing.

To implement \textsc{FindBeta}, we set \( x = [j..n] \), \( y = [1..q - 1] \) and we ask prefix-suffix queries for subsequent values \( d = 1, 3, \ldots, 2^k - 1, \ldots \) until \( d \) exceeds \( \min(|x|, |y|) \). Note that we can terminate the search as soon as a query reports a non-empty answer. Hence, the running time is \( \mathcal{O}(1 + \log \beta) \) if the query is successful (i.e., \( \beta \neq 0 \)) and \( \mathcal{O}(\log n) \) otherwise.

Furthermore, we can expedite the successful calls to \textsc{FindBeta} if we already know that \( \beta \notin \{1, \ldots, \ell\} \). In this case, we can start the search with \( d = \ell + 1 \). Specifically, if \( j \) is not a base reference and belongs to \( \mathcal{T}_j^\ell \) for some \( j' \), we can start from \( d = 2\ell + 1 \) because Lemma 14.2 guarantees that \( \beta \geq \ell + \ell' > 2\ell \).
5 Analysis

Our algorithm computes the longest unbordered factor at each position $i$; position $i$ is a start-reference or it refers to some other position. The correctness of the computed $\text{LUF}[i]$ follows directly from Lemmas 5, 6 and 11.

The analysis of the algorithm running time necessitates probing of the total time consumed by $\text{FINDHOOK}$ and the time spent by $\text{FINDBETA}$ function which, in turn, can be measured in terms of the total size of the stacks of various references.

Lemma 16. The total size of all the stacks used throughout the algorithm is $O(n \log n)$. Moreover, the total running time of the $\text{FINDBETA}$ function is $O(n \log n)$.

Proof. First, we shall prove that any position $p$ belongs to $O(\log n)$ stacks.

By Lemma 14.1, the stack of any reference is a subset of the stack of its parent. Moreover, by Lemmas 14.1 and 15, the stacks of references sharing the same parent are disjoint. A similar argument shows that the stacks of base references are disjoint.

Consequently, the references $j_1 > \ldots > j_s$, whose stacks $S_{j_i}$ contain $p$ form a chain with respect to the parent relation: $j_1$ is a base reference, and the parent of any subsequent $j_i$ is $j_{i-1}$. Let us define $\ell_1, \ldots, \ell_s$ so that $p \in T_j^{\ell_i}$. By Lemma 14.2, for each $1 \leq i < s$, there exist $k_i$ and $l_i \geq \ell_i$ such that $k_i \in T_j^{l_i}$ and $\ell_{i+1} = k_i - p + l_i \geq \ell_i + l_i > 2l_i$. Due to $1 \leq \ell_i \leq n$, this yields $s \leq 1 + \log n = O(\log n)$, as claimed.

Next, let us analyse the successful calls $\beta = \text{FINDBETA}(q, j)$ with $p = q - \beta$. Observe that after each such call, $p$ is inserted to the stack $S_j$ and to the twin set $T_j^{\ell_i}$, i.e., $j = j_i$ and $\beta = \ell_i$ for some $1 \leq i \leq s$. Moreover, if $i > 1$, then $j_i \in T_{j_{i-1}}^{\ell_{i-1}}$, which we are aware of while calling $\text{FINDBETA}$. Hence, we can make use of the fact that $\ell_{i} \notin \{1, \ldots, 2\ell_{i-1}\}$ to find $\beta = \ell_i$ in time $O(\log \frac{\ell_i}{\ell_{i-1}})$. For $i = 1$, the running time is $O(1 + \log \ell_1)$. Hence, the overall running time of successful queries $\beta = \text{FINDBETA}(q, j)$ with $p = q - \beta$ is $O(1 + \log \ell_1 + \sum_{i=2}^{s} \log \frac{\ell_i}{\ell_{i-1}}) = O(1 + \log \ell_s) = O(\log n)$, which sums up to $O(n \log n)$ across all positions $p$.

As far as the unsuccessful calls $0 = \text{FINDBETA}(q, j)$ are concerned, we observe that each such call terminates the enclosing execution of $\text{FINDHOOK}$. Hence, the number of such calls is bounded by $n$ and their overall running time is clearly $O(n \log n)$.

Theorem 17. Given a word $w$ of length $n$, our algorithm solves the Longest Unbordered Factor Array problem in $O(n \log n)$ time with high probability. It can also be implemented deterministically in $O(n \log n \log^2 \log n)$ time.

Proof. Assuming an integer alphabet, the computation of $\text{LSF}_r$ and $\text{LSF}_s$ arrays along with the constant time per position initialisation of the other arrays sum up the preprocessing stage to $O(n)$ time. The running time required for the assignment of the luf for all positions is $O(n)$.

The time spent in construction of the data structure to answer prefix-suffix queries used in $\text{FINDBETA}$ function is $O(n \log n)$ with high probability or $O(n \log n \log^2 \log n)$ deterministic.

Additionally, the total running time of the $\text{FINDHOOK}$ function for all the references, being proportional to the aggregate size of all the stacks, can be deduced from Lemma 16. This has been shown to be $O(n \log n)$ in the worst case, same as the total running time of $\text{FINDBETA}$. The claimed bound on the overall running time follows.

We can also show that the upper bound shown in Lemma 16 is in the worst case tight by designing an infinite family of words that exhibit the worst-case behaviour. We plan to include this construction in the full version of the paper.
6 Final Remark

Computing the longest unbordered factor in $o(n \log n)$ time for integer alphabets remains an open question.

References


Abstract

In real-time systems, in addition to the functional correctness recurrent tasks must fulfill timing constraints to ensure the correct behavior of the system. Partitioned scheduling is widely used in real-time systems, i.e., the tasks are statically assigned onto processors while ensuring that all timing constraints are met. The decision version of the problem, which is to check whether the deadline constraints of tasks can be satisfied on a given number of identical processors, has been known $\mathcal{NP}$-complete in the strong sense. Several studies on this problem are based on approximations involving resource augmentation, i.e., speeding up individual processors. This paper studies another type of resource augmentation by allocating additional processors, a topic that has not been explored until recently. We provide polynomial-time algorithms and analysis, in which the approximation factors are dependent upon the input instances. Specifically, the factors are related to the maximum ratio of the period to the relative deadline of a task in the given task set. We also show that these algorithms unfortunately cannot achieve a constant approximation factor for general cases. Furthermore, we prove that the problem does not admit any asymptotic polynomial-time approximation scheme (APTAS) unless $\mathcal{P} = \mathcal{NP}$ when the task set has constrained deadlines, i.e., the relative deadline of a task is no more than the period of the task.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases multiprocessor partitioned scheduling, approximation factors

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.71

Related Version A full version of the paper is available at [9], https://arxiv.org/abs/1809.04355.
1 Introduction

The sporadic task model has been widely adopted to model recurring executions of tasks in real-time systems [28]. A sporadic real-time task \( \tau_i \) is defined with a minimum inter-arrival time \( T_i \), its timing constraint or relative deadline \( D_i \), and its (worst-case) execution time \( C_i \). A sporadic task represents an infinite sequence of task instances, also called jobs, that arrive with the minimum inter-arrival time constraint. That is, any two consecutive jobs of task \( \tau_i \) should be temporally separated by at least \( T_i \). When a job of task \( \tau_i \) arrives at time \( t \), the job must finish no later than its absolute deadline \( t + D_i \). According to the Liu and Layland task model [27], the minimum inter-arrival time of a task can also be interpreted as the period of the task.

To schedule real-time tasks on multiprocessor platforms, there have been three widely adopted paradigms: partitioned, global, and semi-partitioned scheduling. A comprehensive survey of multiprocessor scheduling in real-time systems can be found in [15]. In this paper, we consider partitioned scheduling, in which tasks are statically partitioned onto processors. This means that all the jobs of a task are executed on a specific processor, which reduces the online scheduling overhead since each processor can schedule the sporadic tasks assigned on it without considering the tasks on the other processors. Moreover, we consider preemptive scheduling on each processor, i.e., a job may be preempted by another job on the processor. For scheduling sporadic tasks on one processor, the (preemptive) earliest-deadline-first (EDF) policy is optimal [27] in terms of meeting timing constraints, in the sense that if the task set is schedulable then it will also be schedulable under EDF. In EDF, the job (in the ready queue) with the earliest absolute deadline has the highest priority for execution. Alternatively, another widely adopted scheduling paradigm is (preemptive) fixed-priority (FP) scheduling, where all jobs released by a sporadic task have the same priority level.

The complexity of testing whether a task set can be feasibly scheduled on a uniprocessor depends on the relations between the relative deadlines and the minimum inter-arrival times of tasks. An input task set is said to have (1) implicit deadlines if the relative deadlines of sporadic tasks are equal to their minimum inter-arrival times, (2) constrained deadlines if the minimum inter-arrival times are no less than their relative deadlines, and (3) arbitrary deadlines, otherwise.

On a uniprocessor, checking the feasibility for an implicit-deadline task set is simple and well-known: the timing constraints are met by EDF if and only if the total utilization \( \sum_{\tau_i \in T} \frac{C_i}{T_i} \) is at most 100\% [27]. Moreover, if every task \( \tau_i \) on the processor is with \( D_i \geq T_i \), it is not difficult to see that testing whether the total utilization is less than or equal to 100\% is also a necessary and sufficient schedulability test. This can be achieved by considering a more stringent case which sets \( D_i = T_i \) for every \( \tau_i \). Hence, this special case of arbitrary-deadline task sets can be reformulated to task sets with implicit deadlines without any loss of precision. However, determining the schedulability for task sets with constrained or arbitrary deadlines in general is much harder, due to the complex interactions between the deadlines and the periods, and in particular is known to be co\(\mathcal{NP}\)-hard or co\(\mathcal{NP}\)-complete [17, 19, 18].

In this paper, we consider partitioned scheduling in homogeneous multiprocessor systems. Deciding if an implicit-deadline task set is schedulable on multiple processors is already \(\mathcal{NP}\)-complete in the strong sense under partitioned scheduling. To cope with these \(\mathcal{NP}\)-hardness issues, one natural approach is to focus on approximation algorithms, i.e., polynomial time algorithms that produce an approximate solution instead of an exact one. In our setting, this translates to designing algorithms that can find a feasible schedule using either (i) faster or (ii) additional processors. The goal, of course, is to design an algorithm that uses the
least speeding up or as few additional processors as possible. In general, this approach is referred to as resource augmentation and is used extensively to analyze and compare scheduling algorithms. See for example [29] for a survey and motivation on why this is a useful measure for evaluating the quality of scheduling algorithms in practice. However, such a measure also has its potential pitfalls as recently studied and reported by Chen et al. [12]. Interestingly, it turns out that there is a huge difference regarding the approximation factors depending on whether it is possible to increase the processor speed or the number of processors. As already discussed in [11], approximation by speeding up is known as the **multiprocessor partitioned scheduling problem**, and by allocating more processors is known as the **multiprocessor partitioned packing problem**. We study the latter one in this paper.

Formally, an algorithm $A$ for the multiprocessor partitioned packing problem is said to have an approximation factor $\rho$, if given any task set $T$, it can find a feasible partition of $T$ on $\rho M^*$ processors, where $M^*$ is the minimum (optimal) number of processors required to schedule $T$. However, it turns out that the approximation factor is not the best measure in our setting (it is not fine-grained enough). For example, it is $\mathcal{NP}$-complete to decide if an implicit-deadline task set is schedulable on 2 processors or whether 3 processors are necessary. Assuming $\mathcal{P} \neq \mathcal{NP}$, this rules out the possibility of any efficient algorithm with approximation factor better than $3/2$, as shown in [11]. (This lower bound is further lifted to 2 for sporadic tasks in Section 5.) The problem with this example is that it does not rule out the possibility of an algorithm that only needs $M^* + 1$ processors. Clearly, such an algorithm is almost as good as optimum when $M^*$ is large and would be very desirable.\(^1\) To get around this issue, a more refined measure is the so-called **asymptotic approximation factor**. An algorithm $A$ has an asymptotic approximation factor $\rho$ if we can find a schedule using at most $\rho M^* + \alpha$ processors, where $\alpha$ is a constant that does not depend on $M^*$. An algorithm is called an asymptotic polynomial-time approximation scheme (APTAS) if, given an arbitrary accuracy parameter $\epsilon > 0$ as input, it finds a schedule using $(1 + \epsilon)M^* + O(1)$ processors and its running time is polynomial assuming $\epsilon$ is a fixed constant.

For implicit-deadline task sets, the multiprocessor partitioned scheduling problem, by speeding up, is equivalent to the Makespan problem [21], and the multiprocessor partitioned packing problem, by allocating more processors, is equivalent to the bin packing problem [20]. The Makespan problem admits polynomial-time approximation schemes (PTASes), by Hochbaum and Shmoys [22], and the bin packing problem admits asymptotic polynomial-time approximation schemes (APTASes), by de la Vega and Lueker [16, 25].

When considering sporadic task sets with constrained or arbitrary deadlines, the problem becomes more complicated. When adopting speeding-up for resource augmentation, the deadline-monotonic partitioning proposed by Baruah and Fisher [3, 4] has been shown to have a $3 - \frac{1}{M}$ speed-up factor in [10], where $M$ is the given number of identical processors. The studies in [2, 11, 1] provide polynomial-time approximation schemes for some special cases when speeding-up is possible. The PTAS by Baruah [2] requires that $\frac{D_{\max}}{D_{\min}}$, $\frac{C_{\max}}{C_{\min}}$, $\frac{T_{\max}}{T_{\min}}$ are constants, where $D_{\max}$ ($C_{\max}$ and $T_{\max}$, respectively) is the maximum relative deadline (worst-case execution time and period, respectively) in the task set and $D_{\min}$ ($C_{\min}$ and $T_{\min}$, respectively) is the minimum relative deadline (worst-case execution time and period, respectively) in the task set. It was later shown in [11, 1] that the complexity only depends on $\frac{D_{\max}}{D_{\min}}$. If $\frac{D_{\max}}{D_{\min}}$ is a constant, there exists a PTAS developed by Chen and Chakraborty [11], which admits feasible task partitioning by speeding up the processors by $(1 + \epsilon)$. The

\(^1\) Indeed, there are (very ingenious) algorithms known for the implicit-deadline partitioning problem that use only $M^* + O(\log^2 M^*)$ processors [25], based on the connection to the bin-packing problem.
Table 1 Summary of the multiprocessor partitioned scheduling and packing problems, unless $\mathcal{P} = \mathcal{NP}$, where $\gamma = \max_{r_i \in \mathcal{T}} \frac{C_i}{\min\{\tau_i, D_i\}}$, $\lambda = \max_{r_i \in \mathcal{T}} \max\{\frac{C_i}{D_i}, 1\}$, and $D_{\text{max}}$ ($D_{\text{min}}$) is the task set’s maximum (minimum) relative deadline. A $^*$ marks results from this paper.

<table>
<thead>
<tr>
<th>Partitioned Scheduling</th>
<th>Implicit Deadlines</th>
<th>Constrained Deadlines</th>
<th>Arbitrary Deadlines</th>
<th>Arbitrary Deadlines (Dependent on $\gamma$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTAS [22]</td>
<td>2.6322-speed up</td>
<td>5-speed up</td>
<td>PTAS [11] for constant $\gamma$</td>
<td></td>
</tr>
<tr>
<td>PTAS (extended from packing) [11]</td>
<td>2.4386-speed up</td>
<td>3-speed up</td>
<td>non-existence of APTAS$^*$</td>
<td></td>
</tr>
<tr>
<td>Partitioned Packing</td>
<td>APTAS [16]</td>
<td>non-existence of APTAS$^*$</td>
<td>non-existence of APTAS$^{(1)}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>33-approximation$^3$, asymptotic $\gamma$-approximation$^4$, non-existence of $(2 - \gamma)$-approximation$^5$</td>
<td></td>
</tr>
</tbody>
</table>

Approach in [11] deals with the multiprocessor partitioned scheduling problem as a vector scheduling problem [7] by constructing (roughly) $(1/\epsilon) \log \frac{D_{\text{max}}}{D_{\text{min}}}$ dimensions and then applies the PTAS of the vector scheduling problem developed by Chekuri and Khanna [7] in a black-box manner. Bansal et al. [1] exploit the special structure of the vectors and give a faster vector scheduling algorithm that is a quasi-polynomial-time approximation scheme (qPTAS) even if $\frac{D_{\text{max}}}{D_{\text{min}}}$ is polynomially bounded.

However, augmentation by allocating additional processors, i.e., the multiprocessor partitioned packing problem, has not been explored until recently in real-time systems. Our previous work in [11] has initiated the study for minimizing the number of processors for real-time tasks. While [11] mostly focuses on approximation algorithms for resource augmentation via speeding up, it also showed that for the multiprocessor partitioned packing problem there does not exist any APTAS for arbitrary-deadline task sets, unless $\mathcal{P} = \mathcal{NP}$. However, the proof in [11] for the non-existence of APTAS only works when the input task set $\mathcal{T}$ has exactly two types of tasks in which one type consists of tasks with relative deadline less than or equal to its period (i.e., $D_i \leq T_i$ for some $\tau_i \in \mathcal{T}$) and another type consists of tasks with relative deadline larger than its period (i.e., $D_j > T_j$ for some $\tau_j \in \mathcal{T}$). Therefore, it cannot be directly applied for constrained-deadline task sets.

For the results, from the literature and also this paper, related to the multiprocessor partitioned scheduling and packing problems, Table 1 provides a short summary.

Our Contributions. This paper studies the multiprocessor partitioned packing problem in more detail. On the positive side, when the ratio of the period of a constrained-deadline task to the relative deadline of the task is at most $\lambda = \max_{r_i \in \mathcal{T}} \max\{\frac{C_i}{D_i}, 1\}$, in Section 3, we provide a simple polynomial-time algorithm with a $2\lambda$-approximation factor. In Section 4, we show that the deadline-monotonic partitioning algorithm in [3, 4] has an asymptotic $\frac{2}{3\gamma}$-approximation factor for the packing problem, where $\gamma = \max_{r_i \in \mathcal{T}} \min\{\frac{C_i}{D_i}, 1\}$. In particular, when $\gamma$ and $\lambda$ are not constant, adopting the worst-fit or best-fit strategy in the deadline-monotonic partitioning algorithm is shown to have an $\Omega(N)$ approximation factor, where $N$ is the number of tasks. In contrast, from [10], it is known that both strategies have a speed-up factor 3, when the resource augmentation is to speed up processors. We also show that speeding up processors can be much more powerful than allocating more processors. Specifically, in Section 5, we provide input instances, in which the only feasible schedule is to run each task on an individual processor but the system requires only one processor with a speed-up factor of $(1 + \epsilon)$, where $0 < \epsilon < 1$.

On the negative side, in Section 6, we show that there does not exist any asymptotic polynomial-time approximation scheme (APTAS) for the multiprocessor partitioned packing problem for task sets with constrained deadlines, unless $\mathcal{P} = \mathcal{NP}$. As there is already an APTAS for the implicit deadline case, this together with the result in [11] gives a complete picture of the approximability of multiprocessor partitioned packing for different types of task sets, as shown in Table 1.
2 System Model

2.1 Task and Platform Model

We consider a set \( T = \{ \tau_1, \tau_2, \ldots, \tau_N \} \) of \( N \) independent sporadic real-time tasks. Each of these tasks releases an infinite number of task instances, called jobs. A task \( \tau_i \) is defined by \( (C_i, T_i, D_i) \), where \( D_i \) is its relative deadline, \( T_i \) is its minimum inter-arrival time (period), and \( C_i \) is its (worst-case) execution time. For a job released at time \( t \), the next job must be released no earlier than \( t + T_i \) and it must finish (up to) \( C_i \) amount of execution before the jobs absolute deadline at \( t + D_i \). The utilization of task \( \tau_i \) is denoted by \( u_i = \frac{C_i}{T_i} \). We consider platforms with identical processors, i.e., the execution and timing property remains no matter which processor a task is assigned to. According to the relations of the relative deadlines and the minimum inter-arrival times of the tasks in \( T \), the task set can be identified to be with (1) implicit deadlines, i.e., \( D_i = T_i \ \forall \ \tau_i \), (2) constrained deadlines, i.e., \( D_i \leq T_i \ \forall \ \tau_i \), or (3) arbitrary deadlines, otherwise. The cardinality of a set \( X \) is denoted by \(|X|\).

In this paper we focus on partitioned scheduling, i.e., each task is statically assigned to a fixed processor and all jobs of the task is executed on the assigned processor. On each processor, the jobs related to the tasks allocated to that processor are scheduled using preemptive earliest deadline first (EDF) scheduling. This means that at each point the job with the shortest absolute deadline is executed, and if a new job with a shorter absolute deadline arrives the currently executed job is preempted and the new arriving job starts executing. A task set can be feasibly scheduled by EDF (or EDF is a feasible schedule) on a processor if the timing constraints can be fulfilled by using EDF.

2.2 Problem Definition

Given a task set \( T \), a feasible task partition on \( M \) identical processors is a collection of \( M \) subsets, denoted \( T_1, T_2, \ldots, T_M \), such that
- \( T_j \cap T_{j'} = \emptyset \) for all \( j \neq j' \),
- \( \bigcup_{j=1}^{M} T_j \) is equal to the input task set \( T \), and
- set \( T_j \) can meet the timing constraints by EDF scheduling on a processor \( j \).

Definition 1. The multiprocessor partitioned packing problem: The objective is to find a feasible task partition on \( M \) identical processors with the minimum \( M \).

We assume that \( u_i \leq 100\% \) and \( \frac{C_i}{T_i} \leq 100\% \) for any task \( \tau_i \) since otherwise there cannot be a feasible partition.

2.3 Demand Bound Function

This paper focuses on the case where the arrival times of the sporadic tasks are not specified, i.e., they arrive according to their interarrival constraint and not according to a pre-defined pattern. Baruah et al. [5] have shown that in this case the worst-case pattern is to release the first job of tasks synchronously (say, at time 0 for notational brevity), and all subsequent jobs as early as possible. Therefore, as shown in [5], the demand bound function \( \text{DBF}(\tau_i, t) \) of a task \( \tau_i \) that specifies the maximum demand of task \( \tau_i \) to be released and finished within any time interval with length \( t \) is defined as

\[
\text{DBF}(\tau_i, t) = \max \left\{ 0, \left[ \frac{t - D_i}{T_i} \right] + 1 \right\} \times C_i.
\]
The exact schedulability test of EDF, to verify whether EDF can feasibly schedule the given task set on a processor, is to check whether the summation of the demand bound functions of all the tasks is always less than \( t \) for all \( t \geq 0 \) [5].

3 Reduction to Bin Packing

When considering tasks with implicit deadlines, the multiprocessor partitioned packing problem is equivalent to the bin packing problem [20]. Therefore, even though the packing becomes more complicated when considering tasks with arbitrary or constrained deadlines, it is pretty straightforward to handle the problem by using existing algorithms for the bin packing problem if the maximum ratio \( \lambda \) of the period to the relative deadline among the tasks, i.e., \( \lambda = \max_{\tau_i \in T} \max \{ \frac{T_i}{D_i}, 1 \} \), is not too large.

For a given task set \( T \), we can basically transform the input instance to a related task instance \( T^\dagger \) by creating task \( \tau_i^\dagger \) based on task \( \tau_i \) in \( T \) such that

\[
T_i^\dagger = D_i, C_i^\dagger = C_i, \text{ and } D_i^\dagger = D_i \quad \text{when } T_i \geq D_i \text{ for } \tau_i, \text{ and}
\]

\[
D_i^\dagger = T_i^\dagger, C_i^\dagger = C_i \text{ and } T_i^\dagger = T_i \quad \text{when } T_i < D_i \text{ for } \tau_i.
\]

Now, we can adopt any greedy fitting algorithms (i.e., a task is assigned to “one” allocated processor that is feasible; otherwise, a new processor is allocated and the task is assigned to the newly allocated processor) for the bin packing problem by considering only the utilization of transformed tasks in \( T^\dagger \) for the multiprocessor partitioned packing problem, as presented in [30, Chapter 8]. The construction of \( T^\dagger \) has a time complexity of \( O(N) \), and the greedy fitting algorithm has a time complexity of \( O(NM) \).

▶ Theorem 2. Any greedy fitting algorithm by considering \( T^\dagger \) for task assignment is a 2\( \lambda \)-approximation algorithm for the multiprocessor partitioned packing problem.

Proof. Clearly, as we only reduce the relative deadline and the periods, the timing parameters in \( T^\dagger \) are more stringent than in \( T \). Hence, a feasible task partition for \( T^\dagger \) on \( M \) processors also yields a corresponding feasible task partition for \( T \) on \( M \) processors. As \( T^\dagger \) has implicit deadlines, we know that any task subset in \( T^\dagger \) with total utilization no more than 100\% can be feasibly scheduled by EDF on a processor, and therefore the original tasks in that subset as well. For any greedy fitting algorithms that use \( M \) processors, using the same proof as in [30, Chapter 8], we get

\[
\sum_{\tau_i \in T^\dagger} \frac{C_i^\dagger}{T_i^\dagger} > \frac{M}{2\lambda}.
\]

By definition, we know that

\[
\sum_{\tau_i \in T} \frac{C_i}{T_i} \geq \sum_{\tau_i^\dagger \in T^\dagger} \frac{C_i^\dagger}{T_i^\dagger} = \frac{M}{2\lambda}. \quad \text{Therefore, any feasible solution for } T \text{ uses at least } \frac{M}{2\lambda} \text{ processors and the approximation factor is hence proved.} \quad \blacktriangleleft
\]

4 Deadline-Monotonic Partitioning under EDF Scheduling

This section presents the worst-case analysis of the deadline-monotonic partitioning strategy, proposed by Baruah and Fisher [4, 3], for the multiprocessor partitioned packing problem. Note that the underlying scheduling algorithm is EDF but the tasks are considered in the deadline-monotonic (DM) order. Hence, in this section, we index the tasks accordingly from the shortest relative deadline to the longest, i.e., \( D_i \leq D_j \) if \( i < j \). Specifically, in the DM partitioning, the approximate demand bound function \( \text{DBF}^*(\tau_i, t) \) is used to approximate Eq. (1), where

\[
\text{DBF}^*(\tau_i, t) = \begin{cases} 0 & \text{if } t < D_i \\ \left( \frac{t - D_i}{\tau_i} + 1 \right) C_i & \text{otherwise.} \end{cases}
\]
Algorithm 1 Deadline-Monotonic Partitioning.

Input: set \( T \) of \( N \) tasks;

1: re-index (sort) tasks such that \( D_i \leq D_j \) for \( i < j \);
2: \( M \leftarrow 1 \), \( T_1 \leftarrow \{ \tau_1 \} \);
3: for \( i = 2 \) to \( N \) do
4: \quad if \( \exists m \in \{ 1, 2, \ldots, M \} \) such that both (3) and (4) hold then
5: \quad \quad choose \( m \in \{ 1, 2, \ldots, M \} \) by preference such that both (3) and (4) hold;
6: \quad \quad assign \( \tau_i \) to processor \( m \) with \( T_m \leftarrow T_m \cup \{ \tau_i \} \);
7: \quad else
8: \quad \quad \( M \leftarrow M + 1 \); \( T_M \leftarrow \{ \tau_i \} \);
9: \quad end if
10: end for
11: return feasible task partition \( T_1, T_2, \ldots, T_M \);

Even though the DM partitioning algorithm in [4, 3] is designed for the multiprocessor partitioned scheduling problem, it can be easily adapted to deal with the multiprocessor partitioned packing problem. For completeness, we revise the algorithm in [4, 3] for the multiprocessor partitioned packing problem and present the pseudo-code in Algorithm 1. As discussed in [4, 3], when a task \( \tau_i \) is considered, a processor \( m \) among the allocated processors where both the following conditions hold

\[
C_i + \sum_{\tau_j \in T_m} DBF^*(\tau_j, D_i) \leq D_i \tag{3}
\]

\[
u_i + \sum_{\tau_j \in T_m} u_j \leq 1 \tag{4}
\]

is selected to assign task \( \tau_i \), where \( T_m \) is the set of the tasks (as a subset of \( \{ \tau_1, \tau_2, \ldots, \tau_{i-1} \} \)), which have been assigned to processor \( m \) before considering \( \tau_i \). If there is no \( m \) where both Eq. (3) and Eq. (4) hold, a new processor is allocated and task \( \tau_i \) is assigned to the new processor. The order in which the already allocated processors are considered depends on the fitting strategy:

- first-fit (FF) strategy: choosing the feasible \( m \) with the minimum index;
- best-fit (BF) strategy: choosing, among the feasible processors, \( m \) with the maximum approximate demand bound at time \( D_i \);
- worst-fit (WF) strategy: choosing \( m \) with the minimum approximate demand bound at time \( D_i \).

For a given number of processors, it has been proved in [10] that the speed-up factor of the DM partitioning is at most 3, independent from the fitting strategy. However, if the objective is to minimize the number of allocated processors, we will show that DM partitioning has an approximation factor of at least \( \frac{N}{4} \) (in the worst case) when the best-fit or worst-fit strategy is adopted. We will prove this by explicitly constructing two concrete task sets with this property. Afterwards, we show that the asymptotic approximation factor of DM partitioning is at most \( \frac{1}{2 - \gamma} \) for packing, where \( \gamma = \max_{\tau_i \in T} \frac{C_i}{\min\{\tau_i, D_i\}} \).

\textbf{Theorem 3.} The approximation factor of the deadline-monotonic partitioning algorithm with the best-fit strategy is at least \( \frac{N}{4} \) when \( N \geq 8 \) and the schedulability test is based on Eq. (3) and Eq. (4).
Packing Sporadic Real-Time Tasks on Identical Multiprocessor Systems

Proof. The theorem is proven by providing a task set that can be scheduled on two processors but where Algorithm 1 when applying the best-fit strategy uses $\frac{N}{2}$ processors. Under the assumption that $K \geq 4$ is an integer, $N$ is $2K$, and $H$ is sufficiently large, i.e., $H \gg K^K$, such a task set can be constructed as:

1. Let $D_1 = 1$, $C_1 = 1/K$, and $T_1 = H$.
2. For $i = 2, 4, \ldots, 2K$, let $D_i = K^{\frac{1}{i-1}}$, $C_i = K^{\frac{1}{i-2}}$, and $T_i = D_i$.
3. For $i = 3, 5, \ldots, 2K - 1$, let $D_i = K^{\frac{1}{i-1}}$, $C_i = K^{\frac{1}{i-2}} - K^{\frac{1}{i-1} - 1}$, and $T_i = H$.

The task set can be scheduled on two processors under EDF if all tasks with an odd index are assigned to processor 1 and all tasks with an even index are assigned to processor 2. On the other hand, the best-fit strategy assigns $\tau_i$ to processor $[\frac{i}{2}]$. The resulting solution uses $K$ processors. Details are in the Appendix in [9].

Theorem 4. The approximation factor of the deadline-monotonic partitioning algorithm with the worst-fit strategy is at least $\frac{N}{2}$ when the schedulability test is based on Eq. (3) and Eq. (4).

Proof. The proof is very similar to the proof of Theorem 3, considering the task set:

1. Let $D_1 = 1$, $C_1 = 1$, and $T_1 = H$.
2. For $i = 2, 4, \ldots, 2K$, let $D_i = K^{\frac{1}{i-1}}$, $C_i = K^{\frac{1}{i-2}}$, and $T_i = D_i$.
3. For $i = 3, 5, \ldots, 2K - 1$, let $D_i = K^{\frac{1}{i-1}}$, $C_i = K^{\frac{1}{i-2}} - K^{\frac{1}{i-1} - 1}$, and $T_i = H$.

Odd tasks are assigned to processor 1 and even tasks to processor 2 the task set is schedulable while $\tau_i$ is assigned to processor $[\frac{i}{2}]$ using the worst-fit strategy. Details are in the Appendix in [9].

Theorem 5. The DM partitioning algorithm is an asymptotic $\frac{2}{1-\gamma}$-approximation algorithm for the multiprocessor partitioned packing problem, when $\gamma = \max_{c_i \in T} \frac{C_i}{\min T_i, D_i}$ and $\gamma < 1$.

Proof. We consider the task $\tau_i$ which is the task that is responsible for the last processor that is allocated by Algorithm 1. The other processors are categorized into two disjoint sets $M_1$ and $M_2$, depending on whether Eq. (3) or Eq. (4) is violated when Algorithm 1 tries to assign $\tau_i$ (if both conditions are violated, the processor is in $M_1$). The two sets are considered individually and the maximum number of processors in both sets is determined based on the minimum utilization for each of the processors. Afterwards, a necessary condition for the amount of processors that is at least needed for a feasible solution is provided and the relation between the two values proves the theorem. Details can be found in the Appendix in [9].

5 Hardness of Approximations

It has been shown in [11, 2] that a PTAS exists for augmenting the resources by speeding up. A straightforward question is to see whether such PTASes will be helpful for bounding the lower or upper bounds for multiprocessor partitioned packing. Unfortunately, the following theorem shows that using speeding up to get a lower bound for the number of required processors is not useful.

Theorem 6. There exists a set of input instances, in which the number of allocated processors is up to $N$, while the task set can be feasibly scheduled by EDF with a speed-up factor $(1 + \epsilon)$ on a processor, where $0 < \epsilon < 1$.

Proof. We provide a set of input instances, with the property described in the statement:

1. Let $D_1 = 1$, $C_1 = 1$, and $T_1 = \frac{(1+\epsilon)^{N-2}}{\epsilon^{N-1}}$. 

Proof. We provide a set of input instances, with the property described in the statement:

1. Let $D_1 = 1$, $C_1 = 1$, and $T_1 = \frac{(1+\epsilon)^{N-2}}{\epsilon^{N-1}}$. 

For any $i = 2, 3, \ldots, N$, let $D_i = \frac{(1+\epsilon)^{i-2}}{i-1}, C_i = D_i$, and $T_i = \frac{(1+\epsilon)^{N-2}}{i-1}$.

Since $C_i = D_i$ for any task $\tau_i$, assigning any two tasks on the same processor is infeasible without speeding up. Therefore, the only feasible processor allocation is $N$ processors and to assign each task individually on one processor. However, by speeding up the system by a factor $1 + \epsilon$, the tasks can be feasibly scheduled on one processor due to $\sum_{i=1}^{N} \frac{dbf(\tau_i, t)}{1+\epsilon} \leq t$ for any $t > 0$. A proof is in the Appendix in [9]. Hence, the gap between these two types of resource augmentation is up to $N$.

Moreover, the following theorem shows the inapproximability for a factor 2 without adopting asymptotic approximation.

Theorem 7. For any $\epsilon > 0$, there is no polynomial-time approximation algorithm with an approximation factor of $2 - \epsilon$ for the multiprocessor partitioned packing problem, unless $P = \mathsf{NP}$.

Proof. Suppose that there exists such a polynomial-time algorithm $A$ with approximation factor $2 - \epsilon$. $A$ can be used to decide if a task set $T$ is schedulable on a uniprocessor, which would contradict the coNP-hardness [17] of this problem. Indeed, we simply run $A$ on the input instance. If $A$ returns a feasible schedule using one processor, we already have a uniprocessor schedule. On the other hand, if $A$ requires at least two processors, then we know that any optimum solution needs $\geq \lceil \frac{M}{2(1-\epsilon)} \rceil = 2$ processors, implying that the task set $T$ is not schedulable on a uniprocessor.

6 Non-Existence of APTAS for Constrained Deadlines

We now show that there is no APTAS when considering constrained-deadline task sets, unless $P = \mathsf{NP}$. The proof is based on an L-reduction (informally an approximation preserving reduction) from a special case of the vector packing problem, i.e., the 2D dominated vector packing problem.

6.1 The 2D Dominated Vector Packing Problem

The vector packing problem is defined as follows:

Definition 8. The vector packing problem: Given a set $V$ of vectors $[v_1, v_2, \ldots, v_N]$ with $d$ dimensions, in which $1 \geq v_{i,j} \geq 0$ is the value for vector $v_i$ in the $j$-th dimension, the problem is to partition $V$ into $M$ parts $V_1, \ldots, V_M$ such that $M$ is minimized and each part $V_m$ is feasible in the sense that $\sum_{v_i \in V_m} v_{i,j} \leq 1$ for all $1 \leq j \leq d$. That is, for each dimension $j$, the sum of the $j$-th coordinates of the vectors in $V_m$ is at most 1.

We say that a subset $V'$ of $V$ can be feasibly packed in an $L$ if $\sum_{v_i \in V'} v_{i,j} \leq 1$ for all $j$-th dimensions. Note that for $d = 1$ this is precisely the bin-packing problem. The vector packing problem does not admit any polynomial-time asymptotic approximation scheme even in the case of $d = 2$ dimensions, unless $P = \mathsf{NP}$ [31].

Specifically, the proof in [11] for the non-existence of APTAS for task sets with arbitrary deadlines comes from an L-reduction from the 2-dimensional vector packing problem as follows: For a vector $v_i$ in $V$, a task $\tau_i$ is created with $D_i = 1$, $C_i = v_{i,2}$, and $T_i = \frac{v_{i,2}}{v_{i,1}}$. However, a trivial extension from [11] to constrained deadlines does not work, since for the transformation of the task set we need to assume that $v_{i,1} \leq v_{i,2}$ for any $v_i \in V$ so that $T_i \geq 1 = D_i$ for every reduced task $\tau_i$. This becomes problematic, as one dimension in the vectors in such input instances for the two-dimensional vector packing problem can be
totally ignored, and the input instance becomes a special case equivalent to the traditional bin-packing problem, which admits an APTAS. We will show that the hardness is equivalent to a special case of the two-dimensional vector packing problem, called the two-dimensional dominated vector packing (2D-DVP) problem, in Section 6.2.

**Definition 9.** The two-dimensional dominated vector packing (2D-DVP) problem is a special case of the two-dimensional vector packing problem with following conditions for each vector \( v_i \in V \):

- \( v_{i,1} > 0 \), and
- if \( v_{i,2} \neq 0 \), then \( v_{i,1} \) is dominated by \( v_{i,2} \), i.e., \( v_{i,2} > v_{i,1} \).

Moreover, we further assume that \( v_{i,1} \) and \( v_{i,2} \) are rational numbers for every \( v_i \in V \).

Here, some tasks are created with implicit deadlines (based on vector \( v_i \) if \( v_{i,2} = 0 \)) and some tasks with strictly constrained deadlines (based on vector \( v_i \) if \( v_{i,2} \) is not 0). However, the 2D-DVP problem is a special case of the two-dimensional vector packing problem, and the implication for \( v_{i,2} > v_{i,1} \) when \( v_{i,2} \neq 0 \) does not hold in the proof in [31]. We note, that the proof for the non-existence of an APTAS for the two-dimensional vector packing problem in [31] is erroneous. However, the result still holds. Details are in the Appendix in [9]. Therefore, we will provide a proper \( L \)-reduction in Section 6.3 to show the non-existence of APTAS for the multiprocessor partitioned packing problem for tasks with constrained deadlines.

### 6.2 2D-DVP Problem and Packing Problem

We now show that the packing problem is at least as hard as the 2D-DVP problem from a complexity point of view. For vector \( v_i \) with \( v_{i,2} > v_{i,1} \), we create a corresponding task \( \tau_i \) with

\[
D_i = 1, \quad C_i = v_{i,2}, \quad T_i = \frac{v_{i,2}}{v_{i,1}}.
\]

Clearly, \( D_i < T_i \) for such tasks. Let \( H \) be a common multiple, not necessary the least, of the periods \( T_i \) of the tasks constructed above. By the assumption that all the values in the 2D-DVP problem are rational numbers and \( v_{i,1} > 0 \) for every vector \( v_i \), we know that \( H \) exists and can be calculated in \( O(N) \). For vector \( v_i \) with \( v_{i,2} = 0 \), we create a corresponding implicit-deadline task \( \tau_i \) with

\[
T_i = D_i = H, \quad C_i = v_{i,1}T_i.
\]

The following lemma shows the related schedulability condition.

**Lemma 10.** Suppose that the set \( T_m \) of tasks assigned on a processor consists of (1) strictly constrained-deadline tasks, denoted by \( T_m^c \), with a common relative deadline \( 1 = D \) and (2) implicit-deadline tasks, i.e., \( T_m \setminus T_m^c \), in which the period is a common integer multiple \( H \) of the periods of the strictly constrained-deadline tasks. EDF schedule is feasible for the set \( T_m \) of tasks on a processor if and only if

\[
\sum_{\tau_i \in T_m^c} C_i \leq 1 \quad \text{and} \quad \sum_{\tau_i \in T_m} u_i \leq 1.
\]
Proof.

Only If. This is straightforward as the task set cannot meet the timing constraint when \( \sum_{\tau_i \in T_m} \frac{C_i}{T_i} > \rho \) or \( \sum_{\tau_i \in T_m} u_i > 1 \).

If. If \( \sum_{\tau_i \in T_m} \frac{C_i}{D} \leq 1 \) and \( \sum_{\tau_i \in T_m} u_i \leq 1 \), we know that when \( t < D \), then \( \sum_{\tau_i \in T_m} DBF(\tau_i, t) = 0 \). When \( D \leq t < H \), we have

\[
\sum_{\tau_i \in T_m} DBF(\tau_i, t) = \sum_{\tau_i \in T_m} \left( \left\lfloor \frac{t-D}{T_i} \right\rfloor + 1 \right) \times C_i 
\leq \sum_{\tau_i \in T_m} C_i + (t-D)u_i 
\leq D + (t-D) = t.
\]

Moreover, with \( \sum_{\tau_i \in T_m} u_i \leq 1 \), we know that when \( t = H \)

\[
\sum_{\tau_i \in T_m} DBF(\tau_i, H) = \sum_{\tau_i \in T_m} \left( \left\lfloor \frac{H-D}{T_i} \right\rfloor + 1 \right) \times C_i 
= \sum_{\tau_i \in T_m} \frac{H}{T_i}C_i + \sum_{\tau_i \in T_m \setminus T_{m_i}} \frac{H}{T_i}C_i = H \left( \sum_{\tau_i \in T_m} u_i \right) \leq H,
\]

where \( =_1 \) comes from the fact that \( \frac{H-D}{T_i} \) is an integer for any \( \tau_i \) in \( T_{m_i} \) and \( T_i > D > 0 \) so that \( \left\lfloor \frac{H-D}{T_i} \right\rfloor + 1 \) is equal to \( \frac{H}{T_i} \).

For any value \( t > H \), the value of \( \sum_{\tau_i \in T_m} DBF(\tau_i, t) \) is equal to \( \sum_{\tau_i \in T_{m_i}} DBF(\tau_i, t-H) + \sum_{\tau_i \in T_m \setminus T_{m_i}} DBF(\tau_i, H) \). Therefore, we know that if \( \sum_{\tau_i \in T_m} C_i \leq 1 \) and \( \sum_{\tau_i \in T_m} u_i \leq 1 \), the task set \( T_m \) can be feasibly scheduled by EDF.

\[\blacksquare\]

**Theorem 11.** If there does not exist any APTAS for the 2D-DVP problem, unless \( \mathcal{P} = \mathcal{NP} \), there also does not exist any APTAS for the multiprocessor partitioned packing problem with constrained-deadline task sets.

Proof. Clearly, the reduction in this section from the 2D-DVP problem to the multiprocessor partitioned packing problem with constrained deadlines is in polynomial time.

For a task subset \( T' \) of \( T \), suppose that \( V(T') \) is the set of the corresponding vectors that are used to create the task subset \( T' \). By Lemma 10, the subset \( T_m \) of the constructed tasks can be feasibly scheduled by EDF on a processor if and only if \( \sum_{\tau_i \in T_m} C_i = \sum_{\tau_i \in V(T_m)} \frac{v_{i,1}}{v_{i,2}} \leq 1 \) and \( \sum_{\tau_i \in T_m} u_i = \sum_{\tau_i \in V(T_m)} v_{i,1} \leq 1 \).

Therefore, it is clear that the above reduction is a perfect approximation preserving reduction. That is, an algorithm with a \( \rho \) (asymptotic) approximation factor for the multiprocessor partitioned packing problem can easily lead to a \( \rho \) (asymptotic) approximation factor for the 2D-DVP problem.

\[\blacksquare\]

### 6.3 Hardness of the 2D-DVP problem

Based on Theorem 11, we are going to show that there does not exist APTAS for the 2D-DVP problem, which also proves the non-existence of APTAS for the multiprocessor partitioned packing problem with constrained deadlines.

**Theorem 12.** There does not exist any APTAS for the 2D-DVP problem, unless \( \mathcal{P} = \mathcal{NP} \).

Proof. This is proved by an L-reduction, following a similar strategy in [31] by constructing an L-reduction from the Maximum Bounded 3-Dimensional Matching (MAX-3-DM), which is MAX SNP-complete [24]. Details are in the Appendix in [9], where a short comment regarding an erroneous observation in [31] is also provided.

\[\blacksquare\]
The following theorem results from Theorems 11 and 12.

▶ **Theorem 13.** There does not exist any APTAS for the multiprocessor partitioned packing problem for constrained-deadline task sets, unless \( P = NP \).

## 7 Concluding Remarks

This paper studies the partitioned multiprocessor packing problem to minimize the number of processors needed for multiprocessor partitioned scheduling. Interestingly, there turns out to be a huge difference (technically) in whether one is allowed faster processors or additional processors. Our results are summarized in Table 1. For general cases, the upper bound and lower bound for the first-fit strategy in the deadline-monotonic partitioning algorithm are both open. The focus of this paper is the multiprocessor partitioned packing problem. If *global scheduling* is allowed, in which a job can be executed on different processors, the problem of minimizing the number of processors has been also recently studied in a more general setting by Chen et al. [14, 13] and Im et al. [23]. They do not explore any periodicity of the job arrival patterns. Among them, the state-of-the-art online competitive algorithm has an approximation factor (more precisely, competitive factor) of \( O(\log \log M) \) by Im et al. [23]. These results are unfortunately not applicable for the multiprocessor partitioned packing problem since the jobs of a sporadic task may be executed on different processors.

### References


Packing Sporadic Real-Time Tasks on Identical Multiprocessor Systems

A Relaxed FPTAS for Chance-Constrained Knapsack

Galit Shabtai
School of Electrical Engineering, Tel Aviv University, Ramat Aviv, Tel Aviv 69978, Israel
galitshabtai@gmail.com

Danny Raz
Faculty of Computer Science, The Technion, Haifa 32000, Israel
danny@cs.technion.ac.il

Yuval Shavitt
School of Electrical Engineering, Tel Aviv University, Ramat Aviv, Tel Aviv 69978, Israel
shavitt@eng.tau.ac.il

Abstract
The stochastic knapsack problem is a stochastic version of the well known deterministic knapsack problem, in which some of the input values are random variables. There are several variants of the stochastic problem. In this paper we concentrate on the chance-constrained variant, where item values are deterministic and item sizes are stochastic. The goal is to find a maximum value allocation subject to the constraint that the overflow probability is at most a given value. Previous work showed a PTAS for the problem for various distributions (Poisson, Exponential, Bernoulli and Normal). Some strictly respect the constraint and some relax the constraint by a factor of $(1 + \epsilon)$. All algorithms use $\Omega(n^{1/\epsilon})$ time. A very recent work showed a “almost FPTAS” algorithm for Bernoulli distributions with $O(poly(n) \cdot \text{quasipoly}(1/\epsilon))$ time.

In this paper we present a FPTAS for normal distributions with a solution that satisfies the chance constraint in a relaxed sense. The normal distribution is particularly important, because by the Berry-Esseen theorem, an algorithm solving the normal distribution also solves, under mild conditions, arbitrary independent distributions. To the best of our knowledge, this is the first (relaxed or non-relaxed) FPTAS for the problem. In fact, our algorithm runs in $\text{poly}(n^{1/\epsilon})$ time.

We achieve the FPTAS by a delicate combination of previous techniques plus a new alternative solution to the non-heavy elements that is based on a non-convex program with a simple structure and an $O(n^2 \log n)$ running time. We believe this part is also interesting on its own right.

2012 ACM Subject Classification Theory of computation → Approximation algorithms analysis, Theory of computation → Stochastic approximation, Theory of computation → Discrete optimization, Theory of computation → Nonconvex optimization

Keywords and phrases Stochastic knapsack, Chance constraint, Approximation algorithms, Combinatorial optimization

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.72

1 Introduction

Stochastic optimization has been studied by a large community since the 1950s. In a stochastic problem the input contains information about distributions rather than concrete values, and the goal is to provide a solution that works well on instances drawn according to the input distributions. For example, one might want to optimize the expected value of certain objective function for inputs drawn from the given input distributions.
An important special case of stochastic optimization is chance-constrained optimization where we want to optimize a target function under the restriction that the probability we violate the constraints is at most some given threshold $p$. For example, Kleinberg, Rabani and Tardos [6], who were among the first to study approximation algorithms for stochastic problems, studied a chance-constrained version of the stochastic knapsack problem, $CCKnapsack$, in the context of bursty connections. In their case the input is information about $n$ items. Item $i$ has value $val_i$ and its size is Bernoulli distributed, i.e., with probability $q_i$ it has size $s_i$ and with probability $1 - q_i$ it has size 0. The distributions of the $n$ items are independent. The input also includes a value $p$ and the knapsack capacity $c$. The goal is to choose a subset of the $n$ items that maximizes the total value subject to the constraint that the overflow probability is at most $p$. Kleinberg et. al. provide a close to linear time $O(\log \frac{1}{p})$-approximation algorithm, by showing a simple reduction to the deterministic case.

Goel and Indyk [4] studied $CCKnapsack$ for several other distributions: Poisson, Exponential and Bernoulli.

- For Poisson they gave a PTAS (Polynomial Time Approximation Scheme). More precisely, given $\epsilon > 0$ the algorithm runs in time $n^{O(1/\epsilon)}$ time and outputs a feasible solution (i.e., a solution where the overflow probability is at most $p$) with value at least $(1 - \epsilon)P^*$, where $P^*$ is the optimal feasible value.

- For the Exponential distributions they obtained a relaxed PTAS, namely, they output an objective value that is no worse than the optimum, but the solution violates the knapsack size and the overflow probability by a factor of $(1 + \epsilon)$.

- For Bernoulli distribution the situation is even worse and they obtain a relaxed QPTAS (Quasi-Polynomial Time Approximation Scheme) algorithm which relaxes the constraints by a factor of $(1 + \epsilon)$, and for a given constant $\epsilon$ runs in quasi-polynomial time in $n$.

Goyal and Ravi [5] present a PTAS for $CCKnapsack$ when item sizes are normally distributed. Their algorithm does not relax the overflow probability constraint nor the capacity constraint. However, it does not give a FPTAS, as the running time of the algorithm is $\Omega(n^{1/\epsilon})$. Later, Bhalgat, Goel and Khanna [1] obtained a PTAS which relaxes both the overflow probability constraint and the capacity constraint and works for any random variable. In a recent work, De [3] showed a “(nearly) FPTAS” for the $CCKnapsack$ with Bernoulli distributions and quasi-FPTAS for $k$-supported random variables, i.e. when all item sizes are supported on a common set of constant size. De [3] also showed a PTAS for hypercontractive random variables, i.e. random variables whose second and fourth moments are within constant factors of each other. Poisson, Gaussian and Exponential random variables are hypercontractive random variables. All three algorithms presented by De [3] relax the overflow probability by an additive $\epsilon$. Table 1 summarizes the above mentioned previous work results.

Goyal and Ravi [5] study the normal distribution case. The normal distribution is particularly interesting since by the central limit theorem the sum of $n$ independent distributions converges to a normal distribution and the Berry Essen theorem gives a concrete bound on the rate of convergence as a function of the first three moments. It is shown in [9] and [8], in a slightly different setting, that an algorithm that solves a chance constrained stochastic problem also works for any $n$ independent distributions, as long as the input distributions respect some mild conditions (e.g., their third moments are reasonable), and this is also true for $CCKnapsack$.

The special case where there are no heavy items, i.e., items whose value is more than $\epsilon$ fraction of the optimal value $P^*$, is particularly interesting, because this is the usual setting for many cloud problems, where there are many services and no single service alone dominates resource demand. In this special case Goyal and Ravi’s algorithm is much faster and runs in $poly(n)$ time (with no dependence on $\epsilon$).
Table 1 Known Results for CCKnapsack.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Distribution</th>
<th>Overflow</th>
<th>Probability</th>
<th>Relaxed Knapsack Capacity</th>
<th>Approximation Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kleinberg et al [6]</td>
<td>Bernoulli</td>
<td>no</td>
<td>no</td>
<td>$O((\log \frac{1}{\epsilon}) P^*)$ polynomial time</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Poisson</td>
<td>no</td>
<td>no</td>
<td>PTAS</td>
<td></td>
</tr>
<tr>
<td>Goel and Indyk [4]</td>
<td>Exponential</td>
<td>yes</td>
<td>yes</td>
<td>PTAS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bernoulli</td>
<td>yes</td>
<td>yes</td>
<td>QPTAS</td>
<td></td>
</tr>
<tr>
<td>Goyal and Ravi [5]</td>
<td>Normal</td>
<td>no</td>
<td>no</td>
<td>PTAS</td>
<td></td>
</tr>
<tr>
<td>Bhalgat et al [1]</td>
<td>any</td>
<td>yes</td>
<td>yes</td>
<td>PTAS</td>
<td></td>
</tr>
<tr>
<td>De [3]</td>
<td>Bernoulli</td>
<td>yes</td>
<td>no</td>
<td>(nearly) FPTAS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$k$-supported</td>
<td>yes</td>
<td>no</td>
<td>quasi-FPTAS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>hypercontractive (Poisson, Gaussian, Exponential, ...)</td>
<td>yes</td>
<td>no</td>
<td>PTAS</td>
<td></td>
</tr>
</tbody>
</table>

Current work Normal no yes FPTAS

We mention that other variants of the problem were studied, e.g., the dynamic knapsack problem [7, 2] where decisions are adaptive and each size is revealed (or realized) only after the decision maker attempts to insert it.

1.1 Previous techniques

Kleinberg et al. [6] show a simple reduction to the deterministic case, by calculating an effective bandwidth value for each item, and then running a greedy algorithm on these deterministic values.

Goel and Indyk [4] proceeded by separating the items to big and small based on whether their value is “large” (which should be appropriately defined) or “small”. On small variables a greedy fractional algorithm is used, and it is easy to see that it has at most one non-integral value. This value is then dropped, and the loss in value is small, because the dropped item is “small”. For “large” items, all candidate sets of large items (and there are at least $2^{1/\epsilon}$ such candidate sets) are checked.

Goyal and Ravi [5] replaced the greedy algorithm with a parametric LP program and showed that the resulting fractional solution has at most two non-integral values. They exploit that for a rounding algorithm that essentially solves correctly (with a small approximation error) the non-heavy elements. For the heavy elements, Goyal and Ravi again try all $n^{1/\epsilon}$ subsets of heavy elements (as there are at most $1/\epsilon$ elements with value $\epsilon P^*$).

Of course, there are more technical issues to be handled. For example, to do the partition to large and small items correctly one needs to know (approximately) the optimal value $P^*$. The solution is to try all approximations to $P^*$ from the set $\{P_{min}, \ldots, P_{min}(1 + \epsilon)^i, \ldots\}$. There must be one $P_i$ in the set such that $P_i \approx P^*$, and the number of such $P_i$ is linear in the input length.
1.2 Our contribution

The results in this paper are two-fold:
- First, we simplify the solution of the light elements (items whose value is at most $\epsilon$ fraction of the optimal value $P^*$) and show a simple non-convex program that has an efficient almost integral solution, and,
- Second, we use it to give a relaxed FPTAS (i.e., an algorithm with running time $\text{poly}(\frac{n}{\epsilon})$ rather than $n^{1/\epsilon}$). We note that this is the first FPTAS algorithm for CCKnapsack, relaxed or not.

We now explain more about these two contributions. First, using a technique from Nikolova [8] we translate the problem on the light elements, to a concrete non-linear (and non-convex) program on $\mathbb{R}^2$. The program is, in fact, a quasi-concave minimization problem, and is minimized on one of the vertices of the polygon of possible solutions. We study this polygon and prove that:
- The polygon has at most $n^2$ vertices,
- These vertices can be easily enumerated, and,
- Each vertex represents an almost integral solution, with at most one non-integral item.

While the base approach is taken from Nikolova [8] the situation here is very different. In [8] the polygon has many (super polynomial) vertices and also it is NP hard to enumerate all the vertices of the polygon. Accordingly, we deviate from the approach taken in [8] and in this paper we use geometric intuition that completely unravel the nature of the polygon in our case. We use the above three properties to construct an efficient algorithm solving CCKnapsack when all the items are not heavy. Specifically, we show:

► Theorem 1. There exists an algorithm for CCKnapsack over normal distributions such that if the value of each element is at most $\epsilon P^*$, where $P^*$ is the optimal feasible value, then the algorithm outputs a feasible integral solution with value at least $(1 - \epsilon)P^*$. The running time of the algorithm is $O(n^2 \log \frac{n}{\epsilon})$.

We remark that Goyal and Ravi’s algorithm [5] also gives a $\text{poly}(n)$ algorithm for the case in which all items are not $\epsilon$-heavy, but our solution is simpler and faster, and we hope that it can also be used in practice.

We now move to our second contribution. As explained above, Goel and Indyk [4] showed a relaxed PTAS for CCKnapsack over several distributions and Goyal and Ravi [5] showed a strict PTAS for the normal distribution. The above algorithms have running time $\Omega(n^{1/\epsilon})$ which indeed allows a PTAS, but is prohibitively large. It is a natural intriguing problem to improve this situation and find a FPTAS whose running time is $\text{poly}(n, f(\epsilon))$ for some function $f$. We show such a result with running time $\text{poly}(\frac{n}{\epsilon})$. We prove:

► Theorem 2. There exists an algorithm solving CCKnapsack over normal distributions that $\epsilon$ approximates the optimum in the relaxed sense, i.e., given an input, it finds a solution such that the overflow probability with a slightly larger capacity $(1 + \epsilon)C$ is at most the specified overflow probability. The running time of the algorithm is $\text{poly}(\frac{n}{\epsilon})$.

The idea is quite natural. We have two basic algorithms for CCKnapsack:
- The algorithm of Theorem 1 that approximates the optimal integral solution for non-heavy elements.
- An exact dynamic programming algorithm that finds an integral solution for Knapsack in time polynomial in the number of partial sums. This algorithm can be easily extended to CCKnapsack.
Suppose we know the optimal integral value $P$. If we divide all items to *light* and *heavy* according to whether their value is smaller than $\epsilon P$ or higher than it, and if we want an $\epsilon$ approximation of the optimal value, we are allowed to lose any constant number of light items, but we are not allowed to lose any heavy item. This leads to the following strategy: On the light items we run the algorithm of Theorem 1. On the Heavy items we cannot miss any single item but we do not mind taking an $\epsilon$ multiplicative approximation. Hence, we round the input values of the heavy items, and run the exact algorithm (that does not lose even a single item) on the rounded heavy items. As there are only few possibilities, the number of partial sums is small, and that part can be efficiently implemented.

There are many technical challenges in implementing the above idea, and our solution is a delicate combination of previous techniques: the multiplicative incremental guessing of parameters, truncation of heavy elements, the dynamic programming for Knapsack and its extension to CCKnapsack and our new reduction to the non-convex problem and its simple structure.

The paper is organized as follows. In Section 2 we show a simple fractional solution to CCKnapsack. In Section 3 we study the non-convex program and the polygon of possible solutions. In [10] we generalize the dynamic programming algorithm of Knapsack to CCKnapsack, and In Section 4 we present our FPTAS with relaxed constraints.

### 2 The Fractional Chance Constrained Knapsack Problem

The Chance Constrained Knapsack Problem for Normal distributions is defined as follows:

<table>
<thead>
<tr>
<th>CCKnapsack: Chance Constrained Knapsack</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> The input to the problem consists of:</td>
</tr>
<tr>
<td>$C$ specifying the knapsack capacity,</td>
</tr>
<tr>
<td>$n$ specifying the number of items available for inclusion in the knapsack,</td>
</tr>
<tr>
<td>$\zeta$ specifying a bound on overflow probability,</td>
</tr>
<tr>
<td>$\epsilon$ - accuracy parameter,</td>
</tr>
<tr>
<td>The size of item $i$, denoted by $X(i)$, is normally distributed with mean $\mu(i)$, and variance $V(i)$ that are given as input. Set $Q(i) = (\mu(i), V(i))$. The distributions $X(i)$ are independent.</td>
</tr>
<tr>
<td>Also, for each item $i$ we are given the value $p(i) &gt; 0$ of that item. A solution $\alpha = (\alpha_1, \ldots, \alpha_n) \in {0, 1}^n$ is feasible if $Pr[\sum_{i=1}^{n} \alpha_iX(i) &gt; C] \leq \zeta$.</td>
</tr>
<tr>
<td><strong>Output:</strong> The output is a vector $\alpha_{\text{out}} = (\alpha_1, \ldots, \alpha_n)$ such that:</td>
</tr>
<tr>
<td>Integrality constraint: $\alpha_i \in {0, 1}$, $\alpha_i = 1$ if item $i$ is selected to be included in the knapsack and $\alpha_i = 0$ otherwise.</td>
</tr>
<tr>
<td>The solution is feasible, and</td>
</tr>
<tr>
<td>Let $P_{\text{out}} = \sum_{i=1}^{n} \alpha_ip(i)$, $P_{\text{OPT}} = \max { \sum_{i=1}^{n} \alpha_ip(i) \mid \alpha_i \in {0, 1}, \alpha \text{ is feasible} }$. We require that $</td>
</tr>
</tbody>
</table>

CCKnapsack is not linear as the overflow probability is not linear. Moreover, the exact problem is clearly NP-hard since its deterministic version, in which each item $X(i)$ takes a single value with probability 1, is the knapsack problem. Therefore, we only ask for an efficient approximation to the problem.

A fractional solution is a feasible vector $\alpha = (\alpha_1, \ldots, \alpha_n)$ with $\alpha_i \in [0, 1]$, dropping the integrality constraint.
Theorem 3. There exists an algorithm that $\epsilon$-approximates the CCKnapsack problem with running time $O(n^2 \log \frac{n}{\epsilon})$, where $n$ is the number of elements. Furthermore, in the fractional solution found, there is at most one fractional item.

We now explain our approach for solving the problem. In the CCKnapsack problem the item sizes are independent and normally distributed. Suppose $\alpha$ is a fractional solution. The size of the solution $\alpha$ is a random variable $X_\alpha = \sum \alpha_i X^{(i)}$ and is normally distributed with mean $\mu_\alpha = \sum_{i=1}^n \alpha_i \mu^{(i)}$ and variance $V_\alpha = \sum_{i=1}^n \alpha_i V^{(i)}$. Also, $\mu_\alpha$ and $V_\alpha$ determine the overflow probability (because the distribution is Normal and is determined by the mean and variance). Hence, we can represent each fractional solution $\alpha$ by the point $(\mu_\alpha, V_\alpha) \in \mathbb{R}^2$ and define the following polygon $\Lambda \subseteq \mathbb{R}^2$ of fractional solutions: $\Lambda = \{Q = \sum_{i=1}^n \alpha_i Q^{(i)} \in \mathbb{R}^2 \mid 0 \leq \alpha_i \leq 1\}$. The algorithm performs a binary search to find an approximate maximum value with overflow probability at most $\zeta$. Each step in the binary search determines whether there exists a feasible solution with a given value $P$. This translates to the question whether the polygon

$$\Lambda_P = \left\{ Q = \sum_{i=1}^n \alpha_i Q^{(i)} \in \mathbb{R}^2 \mid 0 \leq \alpha_i \leq 1, \sum_{i=1}^n \alpha_i P^{(i)} = P \right\},$$

contains a point with overflow probability at most $\zeta$. Equivalently, the problem is whether the minimal overflow probability over the points in the polygon $\Lambda_P$ is at most $\zeta$.

We will see (in Section 3) that the problem of minimizing the overflow probability over the polygon is a quasi-concave minimization problem over a convex body and is seemingly hard. The novelty of the algorithm lies in efficiently solving this problem. This is done by showing (in Section 3) that:

Lemma 4. Fix $P$ and look at the polygon $\Lambda_P$.
1. The minimum overflow probability over points in $\Lambda_P$ is obtained at a vertex of $\Lambda_P$.
2. The polygon $\Lambda_P$ has at most $n^2 - n$ vertices.
3. There exists an algorithm FPBoundary that outputs all vertices in time $O(n^2 \log n)$.

Having that, a simple binary search (over the possible values of $P$) gives Theorem 3 and we give it (along with the correctness proof) in [10].

3 The boundary of the polygon $\Lambda_P$.

The overflow probability of a solution $\alpha = (\alpha_1, \ldots, \alpha_n)$, denoted $OFP(\alpha)$, is,

$$OFP(\alpha) = \Pr\left[\sum_{i=1}^n \alpha_i X^{(i)} > C\right] = \frac{1}{\sqrt{2\pi V_\alpha}} \int_C^\infty e^{-\frac{(x-\mu_\alpha)^2}{2V_\alpha}} dx = 1 - \Phi\left(\frac{C - \mu_\alpha}{\sqrt{V_\alpha}}\right),$$

where $\Phi$ is the cumulative distribution function of the standard Normal distribution. Nikolova shows in [8] that when $C - \mu_\alpha > 0$, $OFP$ is a quasi-concave function (on an $n$ dimensional space $(\alpha_1, \ldots, \alpha_n)$). Also, Nikolova noticed that as $OFP$ depends only on the total mean and variance, when we project the problem to two dimensions (as we did in the previous section) $OFP$ remains quasi-concave. Hence, $OFP$ gets a minimum value over $\Lambda_P$ on a vertex of $\Lambda_P$.

In [10] we prove:

Theorem 5. For every $I_1 = \sum_{i=1}^n \alpha_i Q^{(i)}$ and $I_2 = \sum_{i=1}^n \beta_i Q^{(i)}$ that are adjacent vertices of the polygon $\Lambda_P$, the tuples $\alpha = (\alpha_1, \ldots, \alpha_n)$ and $\beta = (\beta_1, \ldots, \beta_n)$ differ in exactly two elements. Let $k$ and $\ell$ be the indices of these two elements. We call the items $k, \ell$ the active items in vertex $I_1$. 

3.1 Enumerating the polygon vertices

In this section we first show an algorithm PBoundary running in $O(n^2)$ time, and then we introduce a faster algorithm FPBoundary solving the problem in $(n^2 \log n)$ time.

**Algorithm PBoundary**

1. The algorithm first calculates the leftmost vertex, $I_1$, of the polygon $\Lambda_P$ on the mean-variance plane. The point $I_1$ has the minimum mean value among all points in $\Lambda_P$ and therefore it is the leftmost point.

To find $I_1$, sort the input in increasing order of mean to value ratio and re-index it, such that $\frac{\mu_1}{\sigma_1} \leq \ldots \leq \frac{\mu_n}{\sigma_n}$. Denote $I_1 = \sum_{i=1}^{n} \alpha_i Q^{(i)}$. Find the smallest $k$ such that $\sum_{\ell=1}^{k} p^{(\ell)} \geq P$ and set $\alpha_i = 1$ for all $i < k$ and $\alpha_i = 0$ for all $i > k$. Set $\alpha_k$ such that $\sum_{\ell=1}^{k} \alpha_{\ell} p^{(\ell)} = P$. Also, set $E_1 = (0, -1)$.

2. Suppose we have calculated all points $I_1, I_2, \ldots, I_t$ and all vectors $E_1, E_2, \ldots, E_t$. We now calculate the point $I_{t+1}$ and the vector $E_{t+1}$. Denote $I_t = \sum_{i=1}^{n} \alpha_i Q^{(i)}$ and $I_{t+1} = \sum_{i=1}^{n} \beta_i Q^{(i)}$.

   a. **Find direction:** Find a pair of items $(k, \ell) \in [n] \times [n]$, $k \neq \ell$, such that swapping items $k$ and $\ell$ creates the smallest angle with $E_t$. Namely, $\alpha_k > 0$, $\alpha_\ell < 1$ (meaning that we can take more of item $\ell$ and less of item $k$) and the angle between $E_t$ and $Q^{(\ell)} - Q^{(k)}$ is smallest. Also, set $E_{t+1} = Q^{(\ell)} - Q^{(k)}$.

   b. **Edge length:** Set $\beta_\ell = \alpha_k$, $\forall i \notin \{k, \ell\}$. If $\alpha_k p^{(k)} \leq (1 - \alpha_\ell) p^{(\ell)}$, let $\beta_k = 0$ and set $\beta_\ell$ such that $(\beta_\ell - \alpha_k) p^{(\ell)} = \alpha_k p^{(k)}$. Otherwise, let $\beta_\ell = 1$ and set $\beta_k$ such that $(\alpha_k - \beta_k) p^{(k)} = (1 - \alpha_\ell) p^{(\ell)}$.

The algorithm stops when $I_{t+1} = I_1$.

**Claim 6.** The list of points $I_1, I_2, \ldots$ found by PBoundary is the list of all polygon vertices.

**Proof.** Step (1) finds the leftmost point $I_1$ in $\Lambda_P$ which, in particular, is a vertex of $\Lambda_P$. We now want to find the next adjacent vertex, going counterclockwise. According to Theorem 5 two adjacent vertices differ in exactly two items. Among all possible pairs of different items, Step (2a) chooses the pair of indices $(k, \ell) \in [n] \times [n]$ such that inserting item $\ell$ and removing item $k$ creates the smallest angle with the vector $E_1 = (0, -1)$ which is parallel to the variance axis. This sets a direction that equals the direction of the edge leaving $I_1$ in $\Lambda_P$. Step (2b) sets $(\beta_1, \ldots, \beta_n) \in [0, 1]^n$ such that we go in this direction as far as we can while staying in $\Lambda_P$: it either sets $\beta_k$ to 0 or $\beta_\ell$ to 1, which ensure that we go on the chosen direction as far as we can. We therefore stop on the next vertex $I_2$. We then set $E_2 = I_2 - I_1$. Notice that $E_2$ is the *direction* of the edge $(I_1, I_2)$. See Figure 1.
Similarly, suppose we have found the first $t$ vertices $I_1, \ldots, I_t$ for $t > 1$, and $(I_{t-1}, I_t)$ is the last edge found on the boundary so far with direction $E_t = I_t - I_{t-1}$. Again, by Theorem 5 two adjacent vertices differ in exactly two items. Step (2a) chooses a pair of vertices a distinguished one and call it $\Lambda P_{t} \cdot \Lambda P_{t+1}$.

**Corollary 7.** The polygon $\Lambda P$ has at most $n(n - 1)$ vertices.

**Proof.** Let $I_1, \ldots, I_m$ be the vertices in the order found by algorithm PBoundary. Notice that the direction of $(I_j, I_{j+1})$ is $Q^{(t)} - Q^{(k)}$ where $(k, \ell)$ are the active items at vertex $I_j$, and does not depend on the edge length. Thus, there are at most $n(n - 1)$ possible edge directions. As all edges of a convex body in $\mathbb{R}^2$ must have different (directed) directions, we see that the number of vertices is at most $n(n - 1)$.

Algorithm PBoundary takes $O(n^3)$ time, because we have seen that the number of vertices of $\Lambda P$ is at most $n^2$ and if we have found $I_t$, to find the next vertex we go over $n^2$ possible directions. Altogether, the running time is $O(n^4)$. Algorithm PBoundary goes over all $n^2$ possible directions at each step $t$. However, in a convex body that is contained in $\mathbb{R}^2$ the edges have a natural ordering. Suppose the polygon $\Lambda P \subseteq \mathbb{R}^2$ has $m$ vertices. Make any vertex a distinguished one and call it $I_1$. Suppose the other vertices of the polygon are ordered by $I_2, \ldots, I_m$, i.e., $(I_j, I_{j+1} \mod m)$ is an edge of the $\Lambda P$. Let angle$_t$ be the angle between the vectors $I_{t+1} \mod m - I_t$ and $I_2 - I_1$. Then, the angles are monotonically increasing in $t$ until we complete a whole circle and get the zero angle again. Notice that the orientation of a vector is important and the angle between a vector $v$ and $-v$ is $\pi$. See Figure 2.

With that it becomes clear that we do not need to consider all the $n^2$ directions at each time step $t$. Instead we first do a pre-processing step in which we calculate all the $n^2$ directions and sort them by the angle they make with the vector $(0, -1)$ in increasing order. Then, at step $t$ we never look at a direction that we already passed (because the angles are monotonically increasing) and we start the search right after the last entry we have reached in the table. Altogether, our total running time is the table size, reducing the running time from $n^4$ to about $n^2$. We give algorithm FPBoundary and its analysis in [10].

### 3.2 The vertices of polygon $\Lambda P$ are almost integral

**Theorem 8.** Let $I$ be a vertex of $\Lambda P$. Then there is a way to write $I = \sum \alpha_i Q^{(i)}$ with $\alpha = (\alpha_1, \ldots, \alpha_n) \in [0, 1]^n$ such that there is at most one element $k \in [n]$ in which $\alpha_k \notin \{0, 1\}$. 

![Figure 2](image-url)
Proof. Assume that there are at least two elements, \( k \) and \( l \), in which \( \alpha_k, \alpha_l \not\in \{0,1\} \). Let \( \delta_k = \frac{1}{2} \min \{ \alpha_k, 1 - \alpha_k \} \) and \( \delta_l = \frac{1}{2} \min \{ \alpha_l, 1 - \alpha_l \} \). If \( \delta_k p^{(k)} < \delta_l p^{(l)} \), decrease \( \delta_l \) to get \( \delta_k p^{(k)} = \delta_l p^{(l)} \), otherwise, decrease \( \delta_k \) to reach this equality. It is easy to see that both points \( I - \delta_k Q^{(k)} + \delta_l Q^{(l)} \) and \( I + \delta_k Q^{(k)} - \delta_l Q^{(l)} \) are in \( \Lambda_P \), and that the point \( I \) is the mid point of the line connecting them. This contradicts the fact that \( I \) is a vertex. Hence, there is at most one element \( k \in [n] \) in which \( \alpha_k \not\in \{0,1\} \).

\[ \blacksquare \]

▶ Corollary 9. There exists an algorithm for \textit{CKKapsack} such that if the value of each element is at most \( \epsilon P^* \), where \( P^* \) is the optimal feasible value, then the algorithm outputs a feasible integral solution \( I' \) with \( p(I') \geq (1 - \epsilon)P^* \). The running time is \( O(n^2 \log \frac{1}{\epsilon}) \).

\[ \blacksquare \]

3.3 Every fractional point in the polygon is dominated by some integral point with almost the same value

The partial lexicographic order on \( \mathbb{R}^2 \) is \((a, b) \leq (a', b')\) iff \( a \leq a' \) and \( b \leq b' \). We claim:

▶ Lemma 10. For every point \( X \in \Lambda_P \), there exists a vertex \( I \) of \( \Lambda_P \) and an integral point \( A \not\in \Lambda_P \), such that \( A \leq X \), \( A \leq I \) and \( I - A = \gamma Q^{(i)} + \delta Q^{(j)} \) where \( i, j \in [n] \) are the active items at vertex \( I \) and \( 0 \leq \gamma, \delta \leq 1 \). I.e., for every point \( X \) in the polygon \( \Lambda_P \), there exists an integral point \( A \) such that \( A \leq X \) in the lexicographic partial order, and \( A \) is almost a vertex of the polygon differing from some vertex in at most two elements.

\[ \blacksquare \]

Proof. We start at the point \( X \), and vertically go down till we reach the boundary of \( \Lambda_P \) at some point \( Y \). Obviously, \( \mu(Y) = \mu(X) \) and \( V(Y) \leq V(X) \), and hence \( Y \leq X \). Let \( I \) and \( J \) be the two vertices of \( \Lambda_P \) to the left and right of \( Y \), respectively (if \( Y \) is a vertex \( I = Y \)). Notice that \( \mu(I) \leq \mu(Y) = \mu(X) \) and at least one of \( V(I) \leq V(Y) \) or \( V(J) \leq V(Y) \) holds.

According to Theorem (8), \( I \) has a representation \( I = \sum \alpha_i Q^{(i)} \) that has at most one element \( k \in [n] \) in which \( \alpha_k \not\in \{0,1\} \). Let \( i \) and \( j \) be the active items at vertex \( I \).

▶ Claim 11. \( k \in \{i,j\} \).

\[ \blacksquare \]

Proof. Suppose not, i.e. \( \alpha_k \not\in \{0,1\} \) and yet we do a replacement \((i,j)\) in which \( k \) does not participate. Since \( I \) is a vertex we must have \( \alpha_i = 0 \) and \( \alpha_j = 1 \). Now we notice that:

- \( \alpha_k < 1 \) and \( \alpha_i = 1 \), hence \( k \) could have replaced \( i \), and,
- \( \alpha_k > 0 \) and \( \alpha_j = 0 \), hence \( j \) could have replaced \( k \).

The direction of \( k \) replacing \( i \) is \( E_{k,i} = Q^{(k)} - Q^{(i)} \). The direction of \( j \) replacing \( k \) is \( E_{j,k} = Q^{(j)} - Q^{(k)} \). Notice that \( E_{j,i} = E_{j,k} + E_{k,i} \). Hence, \( E_{j,i} \) is inside the parallelogram defined by \( E_{k,i} \) and \( E_{j,k} \), and therefore the algorithm cannot choose the replacement \((i,j)\) since it doesn’t give the smallest angle with the vector \((0,-1)\). A contradiction.

\[ \blacksquare \]

Let \( A \) be the integral point that is received by throwing elements \( i \) and \( j \) from the vertex \( I \) (if they participate in \( I \)), i.e. \( A = \sum_{i \neq \ell \neq j} \alpha_i Q^{(i)} \). Therefore, \( A \leq I \). Also, since vertex \( J \) is received from vertex \( I \) by decreasing element \( i \) and increasing element \( j \), \( J \geq \sum_{\ell \neq i \neq j} \alpha_i Q^{(i)} \). Hence \( A \leq J \). Thus \( A \leq I \) and \( A \leq J \). In particular, \( \mu(A) \leq \mu(I) \leq \mu(Y) \) and also \( V(A) \leq V(I) \) and \( V(A) \leq V(J) \). Therefore, \( V(A) \leq V(Y) \) and hence \( A \leq Y \leq X \).

\[ \blacksquare \]
## The Relaxed FPTAS for CCKnapsack

In this section we present algorithm IntegralRelaxed, which is a relaxed FPTAS for CCKnapsack. The input to IntegralRelaxed is the same as that of CCKnapsack, defined in Section 2. Suppose the optimal solution outputs a set of items \( I^* \) with total value \( P^* \) such that the probability the total size of the items in \( I^* \) exceeds \( C \) is at most \( \zeta \) (\( C \) and \( \zeta \) are inputs to the problem). The output of IntegralRelaxed is a set of items \( I' \) with total value at least \((1 - 5\epsilon)P^* \) and the probability the total size of items in \( I' \) exceeds \((1 + \epsilon)C \) is at most \( \zeta \). Before we explain the algorithm we need a few pre-requisites:

- The algorithm uses as a subroutine a dynamic programming algorithm (described in tails in [10]) that does the following. The input to the dynamic program is a set of triplets \( \{(a(i), b(i), c(i)) \mid i \in S\} \). Define the set of partial sums in each coordinate by \( PS_a = \{\sum_{i \in I} a(i) \mid I \subseteq S\} \), and similarly for \( PS_b \) and \( PS_c \). The algorithm finds all triplets \( \{\sum_{i \in I} (a(i), b(i), c(i)) \mid I \subseteq S\} \), i.e., triplets in \( PS_a \times PS_b \times PS_c \) that can be obtained as the partial sum of the same set \( I \subseteq S \). The algorithm does that by keeping a table whose \( k \)th row records all combinations that can be obtained by the first \( k \) triplets in \( S \). The procedure \( OUT((s_1, s_2, s_3), |S|) \) of the algorithm checks whether \((s_1, s_2, s_3) \) is a feasible triplet, and if so returns a subset \( I \subseteq S \) that obtains it. The running time of the algorithm is polynomial in the number of partial sums.

- We need a truncation operator to truncate heavy elements so that the dynamic programming algorithm runs in polynomial time. We define \([x]_S = \lfloor \frac{x}{T} \rfloor \cdot S\). If \( x \in [0, KS] \), then \([x]_S \) also belongs to \([0, KS]\) but may belong only to a set of \( K \) points which are the first points in consecutive intervals of length \( S \) partitioning \([0, KS]\).

- Define \( OFP_C(\mu, V) = 1 - \Phi(\frac{\mu - V}{\sqrt{V}}) \), and also for \( I = (\mu, v) \) write \( OFP_C(I) = OFP_C(\mu, v) \).

- We use the following fact, when proving that we get a relaxed constraint: For every \( B > 0 \), \( OFP_{B,C}(B\mu, B^2V) = 1 - \Phi(\frac{B\mu}{\sqrt{B^2V}}) = 1 - \Phi(\frac{\mu}{\sqrt{V}}) = OFP_C(\mu, V) \).

- Finally, when we have a sequence \( \{p(i)\}_{i=1}^n \), we think of it as a function \( p(i) = p(i) \) and extend it to sets \( A \subseteq [n] \) by letting \( p(A) = \sum_{a \in A} p(a) \). We do the same for \( \mu, V, p \), etc.

With that IntegralRelaxed does the following. First it guesses four values:

- \( \hat{\mu} \): the guessed total value of the optimal solution.
- \( \hat{\mu}_h \): the guessed value of the heavy elements in the optimal solution.
- \( \hat{V}_h \): the guessed variance of the heavy elements in the optimal solution.

For each such guessed quadruplets, IntegralRelaxed splits the input elements to heavy and light, such that the value of an heavy item is at least \( \epsilon \hat{\mu} \). Then, it truncates the value, the mean and the variance of each heavy element, i.e., \( \bar{p}(i) = [p(i)]_{\hat{\mu}_h/T_1} \), \( \bar{\mu}(i) = [\mu(i)]_{\hat{\mu}_h/T_2} \), \( \bar{V}(i) = [V(i)]_{\hat{V}_h/T_2} \), where \( T_1 = \frac{1}{\epsilon} \) and \( T_2 = \frac{1}{\epsilon^2} n \). The truncated values of the heavy elements are passed as an input to the dynamic programming algorithm, the light elements are solved using Algorithm FPBoundry. More specifically, the algorithm does the following:

---

**IntegralRelaxed(\( \epsilon \))**

- Go over all \( (\hat{\mu}, \hat{\mu}_h, \hat{\mu}_l, \hat{V}_h) \) tuples with \( \hat{\mu}_h \leq \hat{\mu} \) that are in:
  - \( \hat{\mu} \in \{P_{\min}(1 + \epsilon)\mid i \geq 0\} \), \( P_{\min}(1 + \epsilon)^i \leq (1 + \epsilon)P_{\text{total}} \),
  - \( \hat{\mu}_h \in \{P_{\min}(1 + \epsilon)\mid i \geq 0\} \), \( P_{\min}(1 + \epsilon)^i \leq (1 + \epsilon)P_{\text{total}} \),
  - \( \hat{\mu}_l \in \{\mu_{\min}(1 + \epsilon)\mid i \geq 0\} \), \( \mu_{\min}(1 + \epsilon)^i \leq (1 + \epsilon)\mu_{\text{total}} \),
  - \( \hat{V}_h \in \{V_{\min}(1 + \epsilon)\mid i \geq 0\} \), \( V_{\min}(1 + \epsilon)^i \leq (1 + \epsilon)V_{\text{total}} \),

where \( P_{\min} = \min \{p(i)\mid i \in [n]\} \) and \( P_{\text{total}} = \sum_{i \in [n]} p(i) \). Similarly for \( \mu_{\min}, \mu_{\text{total}}, V_{\min}, V_{\text{total}} \).
• For each such \((\hat{p}, \hat{p}_h, \hat{p}_h, \hat{V}_h)\) divide \([n]\) to heavy and light items, \(H = \{i \mid p(i) \geq \hat{c}\}\) and \(L = \{i \mid p(i) < \hat{c}\}\).

• Fix \(T_1 = \frac{1}{\gamma} \) and \(T_2 = \frac{1}{1+\epsilon} \). Run the dynamic programming algorithm presented in [10] on the input \( \{(\overline{p}, \overline{p}_h, \overline{p}_h, \overline{V}_h) = [p(i)]_{p(i)/T_1}, p(i) = [\mu(i)]_{p(i)/T_2}, \overline{V}(i) = [V(i)]_{\overline{V}_h/T_1}\} \), and compute all partial sums. Now we do the following two checks:

1. (Check that \((\hat{p}, \hat{p}_h, \hat{p}_h, \hat{V}_h)\) is heavy-feasible). Check that there exists some partial sum \((\overline{p}, \overline{p}, \overline{V})\) such that \((1-2\epsilon)\overline{p}_h \leq \overline{p} \leq \overline{p}_h, \overline{p} \leq \overline{p}_h, \overline{V} \leq \overline{V}_h\). If there is no such \((\overline{p}, \overline{p}, \overline{V})\) for \((\hat{p}, \hat{p}_h, \hat{p}_h, \hat{V}_h)\) failed. If there is any such \((\overline{p}, \overline{p}, \overline{V})\) let \(H = \text{OUT}((\overline{p}, \overline{p}, \overline{V}), |H|)\).

2. (Check that the light elements can complete a good solution): Let \( \Lambda_{1+\epsilon} \overline{p}_h \) be the polygon defined in Equation (1) using the light items, \(L\), and the target value \(\frac{1}{1+\epsilon} \overline{p} - \overline{p}_h\). Use Algorithm FPBoundry to enumerate the (at most) \(n^2\) vertices of \( \Lambda_{1+\epsilon} \overline{p}_h \). For each vertex \(I\), throw away the two active elements, \(i, j\), to get a set \(L' \subseteq L\) of integral light items (if the polygon is empty, \(L' = \emptyset\)). This step passes if

\[
\frac{\text{OFP}_C(\mu(L')) + \frac{\mu(H')}{1+\epsilon}}{V(L')} + \frac{V(H')}{1+\epsilon} \leq \frac{\zeta}{\epsilon}.
\]

• Let \(\overline{p}\) be the largest value for which steps 1 and 2 passed for some \((\hat{p}_h, \hat{p}_h, \hat{V}_h)\). Suppose we used \((\overline{p}, \overline{p}, \overline{V})\) and \(H'\) in step 1 and \(L'\) in step 2 when accepting \((\hat{p}_h, \hat{p}_h, \hat{V}_h)\). Return \(H' \cup L'\).

Lemma 12. Let \(\epsilon, T_1, T_2\) be as defined before. Let \(\lambda^*\) be an optimal feasible integral solution with value \(P^*\), mean \(\mu\) and variance \(V^*\). Let \(\overline{p}\) be such that \(P^* \leq \overline{p} \leq (1+\epsilon)P^*\). Define \(H = \{i \mid p(i) \geq \hat{c}\}\) and \(L = \{i \mid p(i) < \hat{c}\}\). Denote \(H^* = \lambda^* \cap H\) and \(L^* = \lambda^* \cap L\). Let \(\hat{p}_h, \hat{p}_h, \hat{V}_h\) be such that \(p(H^*) \leq \hat{p}_h \leq (1+\epsilon)p(H^*), \mu(H^*) \leq \hat{p}_h \leq (1+\epsilon)\mu(H^*)\) and \(V(H^*) \leq \hat{V}_h \leq (1+\epsilon)V(H^*)\). Then, IntegralRelaxed(\(\epsilon\)) accepts \((\overline{p}_h, \overline{p}, \overline{V}, \overline{V})\) and the value of the associated set is at least \((1-5\epsilon)P^*\).

Proof. One of the solutions the dynamic programming algorithm generates is \((\overline{p} = p(H^*), \overline{p} = p(H^*), \overline{V} = \overline{V}(H^*))\). It is clear that \(\overline{p} = p(H^*) \leq p(H^*) \leq \hat{p}_h\). Similarly, \(\overline{p} \leq \hat{p}_h\) and \(\overline{V} \leq \hat{V}_h\).

Also, we first notice that for every \(i \in H\) we have

\[
\overline{p}(i) = [p(i)]_{\overline{p}_h/T_1} \geq p(i) - \frac{\hat{p}_h}{T_1} = p(i) - \epsilon^2 \hat{p}_h \geq p(i) - \epsilon \hat{p}_h \geq (1-\epsilon)p(i),
\]

because \(p(i) \geq \hat{c} \geq \hat{c} \hat{p}_h\) (as \(i \in H\)). Therefore, \(\overline{p} = p(H^*) \geq (1-\epsilon)p(H^*) \geq \frac{1}{1+\epsilon} \hat{p}_h \geq (1-2\epsilon)\hat{p}_h\). Hence the check at step 1 passes. Let \(H' = \text{OUT}((\overline{p}, \overline{p}, \overline{V}), |H|)\).

Next the algorithm does the check at step 2. We first notice that \(L^*\) has value \(p(L^*) = p(L^*) - p(H^*), P^* \geq \hat{p}_h \geq \hat{p}_h, \hat{p}_h \), and therefore the polygon \(\Lambda_{1+\epsilon} \overline{p}_h\) is not empty and there exists some \(X\) in the polygon that is supported over elements from \(L^*\), i.e., \(X = \sum_{i \in L} \alpha_i Q(i) \leq L^*\). By Lemma (10) there exists a vertex \(I\) and an integral point \(L'\) such that \(L' \leq X\), and \(I - L' = \gamma Q(i) + \delta Q(j)\) for \(i\) and \(j\) that are the active items in vertex \(I\) and \(0 \leq \gamma, \delta \leq 0\).

When the algorithm goes over all the vertices in \(\Lambda_{1+\epsilon} \overline{p}_h\), it also checks \(I\), and it computes the integral solution \(L'\) (which is the vertex \(I\) with the active items removed), and when checking \(L'\) we have \(\mu(L') \leq \mu(X) \mu(L') \leq V(L) \leq V(L') \). Notice that \(p(H^*) \leq p(H^*) + nQ(i) \leq p(H^*) + n \frac{(1+\epsilon)p(H^*)}{T_2} \leq (1+\epsilon)p(H^*)\). Similarly, \(V(H^*) \leq V(H^*)\). Together,

\[
\frac{\text{OFP}_C(\mu(L') + \frac{\mu(H^*)}{1+\epsilon})}{V(L')} + \frac{V(H^*)}{1+\epsilon} \leq \text{OFP}_C(\mu(L') + \mu(H^*), V(L') + V(H^*)) = \text{OFP}_C(\mu(L'), V(L')) \leq \zeta.
\]
Thus, when the algorithm gets to check vertex $I$ (and the algorithm checks all vertices $I$) the algorithm accepts and returns the solution $H' \cup L'$. Finally, notice that

$$p(H' \cup L') = p(H') + p(L') \geq P(H') + \frac{1}{1+\epsilon} \hat{p} - \hat{p}_n - 2 \epsilon \hat{p} \geq (1 - \epsilon) \hat{p} - \hat{p}_n - 2 \epsilon \hat{p}$$

$$\geq (1 - 2 \epsilon) \hat{p}_n + (1 - 3 \epsilon) \hat{p} - \hat{p}_n = (1 - 3 \epsilon) \hat{p} - 2 \epsilon \hat{p}_n \geq (1 - 5 \epsilon) \hat{p} \geq (1 - 5 \epsilon) P^*$$

\textbf{Lemma 13.} If algorithm $\text{IntegralRelaxed}(\epsilon)$ returns the set $I'$, then $\text{OFP}_{(1+\epsilon)}(I') \leq \zeta$.

\textbf{Proof.} Suppose the algorithm accepts and returns the integral set $I' = H' \cup L'$. As the algorithm passed both checks, we know that $\text{OFP}_{(1+\epsilon)}(\mu(H'), V(L') + V(H')) \leq \zeta$. Hence,

$$\text{OFP}_{(1+\epsilon)}(\mu(I'), V(I')) = \text{OFP}_{(1+\epsilon)}(\mu(L') + \mu(H'), V(L') + V(H'))$$

$$\leq \text{OFP}_{(1+\epsilon)}((1 + \epsilon) \mu(L') + \frac{\mu(H')}{1 + \epsilon}, (1 + \epsilon)^2 (V(L') + \frac{V(H')}{1 + \epsilon}))$$

$$= \text{OFP}(\mu(L') + \frac{\mu(H')}{1 + \epsilon}, V(L') + \frac{V(H')}{1 + \epsilon}) \leq \zeta.$$

In [10] we prove:

\textbf{Lemma 14.} $\text{IntegralRelaxed}(\epsilon)$ takes $\tilde{O}(\frac{\text{len} \cdot n^6}{\epsilon^5})$ time, where $\text{len}$ is the input length.

\section*{References}


Covering Clients with Types and Budgets

Dimitris Fotakis
Yahoo Research-New York, USA & National Technical University of Athens, Greece
dfotakis@oath.com

Laurent Gourvès
Université Paris-Dauphine, PSL University, CNRS, LAMSADE, 75016 Paris, France
laurent.gourves@dauphine.fr

Claire Mathieu
CNRS, France
clairemmathieu@gmail.com

Abhinav Srivastav
ENS Paris & Université Paris-Dauphine, France
abhinavsriva@gmail.com

Abstract
In this paper, we consider a variant of the facility location problem. Imagine the scenario where facilities are categorized into multiple types such as schools, hospitals, post offices, etc. and the cost of connecting a client to a facility is realized by the distance between them. Each client has a total budget on the distance she/he is willing to travel. The goal is to open the minimum number of facilities such that the aggregate distance of each client to multiple types is within her/his budget. This problem closely resembles the set cover and \( r \)-domination problems. Here, we study this problem in different settings. Specifically, we present some positive and negative results in the general setting, where no assumption is made on the distance values. Then we show that better results can be achieved when clients and facilities lie in a metric space.

2012 ACM Subject Classification Theory of computation → Packing and covering problems, Theory of computation → Facility location and clustering

Keywords and phrases Facility Location, Geometric Set Cover, Local Search

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.73

Funding PSL Project MULTIFAC

1 Introduction
Consider the problem of opening a set of facilities, such as public service centres, in a city such that all clients (people living in the city) are within a pre-specified distance of some facility. The objective here is to open the minimum number of facilities. This problem closely resembles the \( r \)-dominating set problem where given a metric space \((V,d)\) and a distance threshold \(r\), the goal is to find a minimum-size set \(M\) of points such that every point in \(V\) is within distance \(r\) to some point in \(M\). This is a special case of the classical set cover problem and can be approximated within a factor of \(1 + \ln |V|\). In the Euclidean plane, however, a polynomial time approximation scheme follows from the results on geometric covering problems of Hochbaum and Maass [17].

In this paper, we study the generalization of the \(r\)-dominating set problem with different types of covering points. Consider the setting where facilities can be categorized into multiple types, such as schools, hospitals, post offices, etc., and the cost of connecting a client to a
facility can be realized by the distance between them. Each client has a total budget on the distance he/she is willing to travel. As in set cover and \( r \)-dominating set problems, the goal is to open the minimum number of facilities so that the aggregate distance of each client to the nearest facilities of all types is within his/her budget. Intuitively, each client is willing to accept tradeoffs among his/her distance to different facility types. Facility location with multiple types has been previously studied in [15, 3]. Hajiaghayi et al. [15] considered a variant of the \( k \)-median problem with two facility sets (red and blue), where we can open at most \( k_r \) red and \( k_b \) blue facilities. As opposed to our problem, each client is assigned to a single nearest facility that can be either red or blue. The goal is to minimize the total distance of the clients to their facility.

Problem Definition. We are given a set \( F \) of \( m \) facilities that are partitioned into \( L \) types \( F_1, F_2, \ldots, F_L \), and a set \( C \) of \( n \) clients, each with a budget \( B_j \) for \( j \in \{1, \ldots, n\} \). We assume that \( L \) is a constant and that \( m = O(n^c) \), for some fixed constant \( c \). Moreover, we are given a distance matrix \( D \) of size \(|F| \times |C|\), where each element \( d_{ij} \) represents the distance between facility \( i \) and client \( j \). We say that a client \( j \) is served or covered by a type-\( \ell \) facility \( i \), if \( i \) is the nearest open facility of type-\( \ell \) to \( j \). Furthermore, we say that \( j \) has a service cost or covering cost of \( d_{ij} \) for facilities of type-\( \ell \). The total service (or covering) cost of \( j \) is the sum of \( j \)'s service costs over all types. Our goal is to compute a set \( S \) of facilities of minimum cardinality such that each client \( j \) is served by one open facility of each type in \( S \) and the total service cost of \( j \) is at most \( B_j \). We refer to this problem as FLT that is, Facility Location with Types.

In this paper, we present bi-criteria approximations of FLT problem in different settings. Let \( OPT \) denote the number of facilities opened by a fixed optimal solution. We say that a solution \( S' \subseteq F \) is \((\alpha, \beta)\)-approximate iff the number of facilities opened in \( S' \) is at most \( \alpha \cdot OPT \) and the total service cost of each client \( j \) with respect to \( S' \) is at most \( \beta B_j \). As usual, an algorithm \( A \) is \((\alpha, \beta)\)-approximate if it outputs an \((\alpha, \beta)\)-approximate solution for every instance.

Related Work. The FLT problem with \( L = 1 \) corresponds to the set cover problem, where given a set of elements \( U \) and a collection \( S \) of subsets of \( U \), the aim is to choose a minimum number of sets in \( S \) such that every element in \( U \) is covered. The analogy with FLT is straightforward: \( U \) and \( S \) correspond to the set of clients \( C \) and the set of the facilities \( F \), respectively such that a client \( j \) is contained in the set corresponding to facility \( i \) if \( d_{ij} \leq B_j \).

It is known that the set cover problem admits \( \Delta \)-approximation algorithms where \( \Delta = 1 + \frac{1}{2} + \cdots + \frac{1}{n} \leq (1 + \ln n) \) and \( f \) is the maximum frequency of any element in \( U \).

Dimur and Steurer [9] proved that it is NP-hard to approximate the set cover problem within a ratio of \((1 - \epsilon) \ln n \), for any \( \epsilon > 0 \). Another problem equivalent to the set cover problem is the hitting-set problem, where given a set of element \( U \) and a collection \( S \) of subsets of \( U \), the aim is to choose the minimal set of elements \( P \) in \( U \) such that \( P \cap S \neq \emptyset \), for all \( S \in S \). In a general setting, all results for the set cover problem extend to the hitting-set problem.

Surprisingly, the hitting-set problem admits a better approximation ratio in \( \mathbb{R}^2 \) (also called geometric hitting-set or GHS). Mustafa and Ray [20] showed that a simple local search algorithm is a PTAS for the problem where elements in \( U \) and subsets in \( S \) correspond to points and pseudo-disks, respectively, in \( \mathbb{R}^2 \). As such, there are no fully polynomial approximation scheme for this problem unless \( \mathcal{NP} = \mathcal{P} \) [14]. The FLT problem in \( \mathbb{R}^2 \) is closely related to the problem of covering a set of points with ellipses. For a special case of this problem where the ellipses are axis-parallel, Efrat et al. [11] presented an \( O(n^* \log n^*) \) approximation, where \( n^* \) is the size of an optimal cover.
Another problem related to FLT problem is the red-blue set cover problem [8]. Here, elements in \( U \) are partitioned into two sets: red set \( R = \{ r_1, r_2, \ldots, r_k \} \) and blue set \( B = \{ b_1, b_2, \ldots, b_l \} \). The objective is to find a collection of subsets in \( S \) such that all blue elements are covered and the number of red elements covered is minimized. Carr et al. [8] showed that red-blue set cover problem cannot be approximated within a factor of \( O(2^{\log^{-1} n}) \) for any \( \epsilon > 0 \), where \( n' = |S|^4 \) (also see [12] for a similar inapproximability result). Further, Carr et al. [8] showed that red-blue set cover admits an \( O((cp)^{1-1/c}\log\rho) \)-approximation, where \( \rho = |S| \) and \( c \geq |S \cap R| \) for all \( S \in \mathcal{S} \).

FLT requires that every client is covered by \( L \) facilities, which is reminiscent of the set multi-cover problem [4] and the fault-tolerant facility location problem (FT-FL in short) [18, 21]. Every element (resp. client) has a demand which is a lower bound on the number of sets where the element must appear (resp. a lower bound on the number of facilities to which the client is assigned). However, unlike FLT, the sets (resp. facilities) are not categorized and a coverage with one set (resp. facility) of each category is not imposed. FLT also bears some remote resemblance to multilevel facility location problems, where facilities are partitioned into \( k \) levels and each client must travel to a facility at level \( k \) through a path that goes through one facility at each level \( 1, \ldots, k \) (see e.g., [1, 6] and the references therein). Unlike FLT, in multilevel facility location, the clients move from lower to higher levels and there is no budget on the total length of the path.

The literature contains various aggregate functions for capturing the distance between a client and its \( L \) covering facilities: maximum distance, sum of the distances, or more generally with the use of an ordered weighted average [22]. In this article we consider the sum, like for FT-FL. However, the clients’ total covering costs are part of the objective function in FT-FL, whereas they are treated as constraints in FLT, i.e. client \( j \)’s total service cost should not exceed a prescribed budget \( B_j \).

### 1.1 Our Contribution

To the best of our knowledge, the approximability of covering problems with multiple types and a constraint on the combined “quality” of each client’s covering has not been studied before. In this work, we give an almost complete picture of the approximability of FLT for both general and metric instances. For general instances, no specific assumption is made on the distance values in \( D \). For metric instances, we assume that the values in \( D \) satisfy the triangle inequality. We obtain stronger results for Euclidean instances, where the clients and the facilities lie in either \( \mathbb{R} \) or \( \mathbb{R}^2 \) and the Euclidean distance is used. Many of our results (especially those for general instances) can be extended to non-uniform facility opening costs.

**General Instances.** For general instances, we almost match the approximability of set cover, by slightly violating the budget constraint. If we insist on satisfying the budget constraint, FLT becomes difficult to approximate even for \( L = 2 \). More specifically, in Section 2, we obtain the following results:

1. A greedy algorithm achieves an approximation ratio of \( n(1/\sqrt{\log n}) \). This matches the classical result for set cover when \( L = 1 \). We also present an example showing that our analysis is almost tight.
2. By extending the greedy algorithm for set cover, we obtain bi-criteria approximation algorithms with approximation guarantees of \( (\mathbb{H}_n, L) \) and \( (2\mathbb{H}_n, L - 1 + 1/L) \) for FLT.
3. By generalizing the randomized rounding algorithm for the set cover problem, we obtain a bi-criteria approximation of \( O((\log n/c), 1 + \epsilon) \). So, we can achieve an asymptotically best possible logarithmic approximation, if we violate the budget constraint by a small constant factor. This result holds for non-uniform facility costs as well.
4. We propose a nontrivial generalization of the frequency parameter used for SET COVER. Formally, for \( L = 2 \), we introduce a parameter \( \psi \), which is always bounded from above by the maximum number of facility pairs that can serve a client. Then, we obtain an LP-based \( \psi \)-approximation algorithm for \( L = 2 \) which satisfies the budget constraint.

5. If we insist on satisfying the budget constraint, one should not expect much better approximation guarantees. Using a transformation from SYMMETRIC LABEL COVER [10], we show that FLT cannot be approximated within a ratio of \( O(2^{\log^{1/2-\epsilon} L}) \), for any \( \epsilon > 0 \), unless \( \mathcal{NP} \) is in quasipolynomial time. Here, \( \tau \) is no less than the maximum number of facility pairs that can cover any client.

**Metric Instances.** FLT becomes significantly easier to approximate in metric instances. This is especially true for Euclidean instances. More formally, in Section 3, we obtain the following results:

1. We show that a natural greedy algorithm achieves a bi-criteria guarantee of \((1, 3L)\), if the distance matrix \( D \) satisfies the triangle inequality.
2. By extending the dynamic programming algorithm for \( k \)-median on the real line, we show that FLT can be solved optimally in polynomial time for linear instances. This result can be extended to non-uniform facility costs (with a slightly different recursion though).
3. By extending the techniques of Mustafa and Ray [20], we obtain bi-criteria approximation algorithms with guarantees of \((1 + \epsilon, L)\) and \((2 + \epsilon, L - 1 + 1/L)\) for instances on \( \mathbb{R}^2 \).
4. Our main result is that FLT on the Euclidean plane admits a bi-criteria polynomial-time approximation scheme, with an approximation guarantee of \((1 + \epsilon, 1 + \epsilon)\), if all clients have a uniform budget \( B \).

## 2 General Instances

Recall that for general instances of the FLT problem no specific assumptions are made on the distances in \( D \). The following lemma presented in [8] can be adapted to the FLT problem.

**Lemma 1.** RED-BLUE SET COVER has a \( \mathcal{O}((\psi\pi)^{1-1/c}\log \rho) \)-approximation algorithm when \( \forall S \in \mathcal{S}, |S \cap R| \leq c \) where \( \rho = |\mathcal{S}| / 8 \).

If we restrict the type of facilities to 2 that is, \( L = 2 \) the Lemma 1 implies that there exists an \( \mathcal{O}((\sqrt{n}2n) \log n) \)-approximation algorithm. Below, we present a simple greedy algorithm that achieves an approximation ratio of \( \sqrt{n}H_n \) for 2 types of facilities.

### 2.1 Deterministic Algorithm

In Algorithm 1, we say that a client \( c \) is covered by a tuple \((i_1, \ldots, i_L)\) if \( \sum_{\ell=1}^L d_{c,i_\ell} \leq B_c \). Note that when considering \((i_1, \ldots, i_L)\), the algorithm does not take into account the clients of \( U \) that are covered by a tuple consisting of some facilities in \((i_1, \ldots, i_L)\) and some facilities that are already present in \( S \) and had been selected in previous rounds.

**Theorem 2.** Algorithm 1 is an \( (n (\frac{\psi \pi}{\tau})^{1/L}) \)-approximation algorithm for the FLT problem.

**Proof.** Fix an instance and its optimal solution \( Y \). Suppose \( |Y \cap F_\ell| = a_\ell, \forall \ell \in [L] \). Then, \( |Y| = \sum_{\ell=1}^L a_\ell \). For each tuple of \( L \) facilities \((i_1, \cdots, i_L)\) \in \((Y \cap F_1) \times \cdots \times (Y \cap F_L)\), create a bag \( B(i_1, \cdots, i_L) \). Each client \( c \) is put in exactly one bag \( B(i_1, \cdots, i_L) \) such that \( \sum_{\ell=1}^L d_{c,i_\ell} \leq B_c \). Break ties arbitrarily for the clients who can be placed in several bags. The \( \prod_{\ell=1}^L a_\ell \) bags form a partition of \( C \).
Theorem 2 gives a bi-criteria approximation algorithm. Next, we present another incomparable bi-criteria approximation algorithm. The exact proposition and proof is omitted due to space constraints.

Algorithm 1:

1. Initialize $S \leftarrow \emptyset$ and $U \leftarrow C$
2. while $U \neq \emptyset$ do
   3. Choose $(i_1, \ldots, i_L) \in F_1 \times \cdots \times F_L$ such that the number of clients in $U$ covered by $(i_1, \ldots, i_L)$ is maximized
   4. Add $\{i_1, \ldots, i_L\}$ to $S$ and remove from $U$ the clients covered by $(i_1, \ldots, i_L)$
5. return $S$

We repeatedly use the arithmetic-geometric means inequality: $\sum_{\ell=1}^{L} a_{\ell} \geq L (\prod_{\ell=1}^{L} a_{\ell})^{1/L}$.

Let $X$ denote the solution output by Algorithm 1. We claim that

$$|X| \leq L (\prod_{\ell=1}^{L} a_{\ell}) H_n. \quad (1)$$

To see this, observe the choices made by Algorithm 1 on the bags defined above. The first greedy choice is at least as good as covering the largest bag (i.e. selecting its corresponding facilities). Afterwards, update the bags by removing the clients currently covered by the partial greedy solution. The next choice is again, at least as good as covering the largest bag, and so on. Because there are $\prod_{\ell=1}^{L} a_{\ell}$ bags, the optimal solution uses at most $\prod_{\ell=1}^{L} a_{\ell}$ sets to cover the $n$ clients. As for SET COVER, Algorithm 1 needs at most $\sum_{\ell=1}^{L} a_{\ell} H_n$ rounds to cover all the clients, each round requiring at most $L$ new facilities.

Suppose $\prod_{\ell=1}^{L} a_{\ell} \leq \frac{n}{H_n}$. It follows from $|Y| = \sum_{\ell=1}^{L} a_{\ell}$ and (1) that the approximation ratio is at most $\frac{L (\prod_{\ell=1}^{L} a_{\ell}) H_n}{\sum_{\ell=1}^{L} a_{\ell}}$. Combining this with the arithmetic-geometric means inequality, we obtain that $\prod_{\ell=1}^{L} a_{\ell} H_n \leq (\prod_{\ell=1}^{L} a_{\ell})^{1-1/L} H_n$. Using the fact that $\prod_{\ell=1}^{L} a_{\ell} \leq \frac{n}{H_n}$, we get that $(\prod_{\ell=1}^{L} a_{\ell})^{1-1/L} H_n \leq \left( \frac{n}{H_n} \right)^{1-1/L} H_n = n^{1-1/L} (H_n)^{1/L}$.

Now suppose $\prod_{\ell=1}^{L} a_{\ell} > \frac{n}{H_n}$. We have $|X| \leq Ln$ because in the worst case, each client requires its own tuple of $L$ facilities. The approximation ratio is at most $\frac{L n}{\sum_{\ell=1}^{L} a_{\ell}}$. Using the arithmetic-geometric means inequality, we get that $\sum_{\ell=1}^{L} a_{\ell} \leq \frac{n}{H_n}$. Thus, $\prod_{\ell=1}^{L} a_{\ell} H_n \leq \left( \frac{n}{H_n} \right)^{1-1/L} H_n$ and $\prod_{\ell=1}^{L} a_{\ell} > \frac{n}{H_n}$ raised to the power of $1/L$ to get that $\prod_{\ell=1}^{L} a_{\ell} H_n \leq n^{1-1/L} (H_n)^{1/L}$.

An almost tight instance: Take a positive integer $t$ and create a set of $n = t^L$ clients $\{1, \ldots, t\}^L$. Each client is associated with a vector $\vec{c} \in \{1, \ldots, t\}$. The client with vector $\vec{c}$ can be covered by two separate sets of facilities: $(f_{\vec{c}}, \ldots, f_{\vec{c}})$ and $(g_{\vec{c}}, \ldots, g_{\vec{c}})$. The optimum takes the “$f$” facilities (there are $Lt$ such facilities) whereas the greedy algorithm can pick the “$g$” facilities (there are $Lt$ such facilities). For the described family of instances, Algorithm 1 returns a $t^{L-1} = n^{1-1/L}$-approximate solution.

2.2 Bi-criteria Approximations

Theorem 2 gives a bi-criteria $(n \left( \frac{H_n}{n} \right)^{1/L}, 1)$-approximation result for FLT problem. The simple strategy of solving $L$ separate instances of SET COVER provides a bi-criteria $(H_n, L)$-approximation algorithm. The exact proposition and proof is omitted due to space constraints. Next, we present another incomparable bi-criteria approximation algorithm.
Proposition 3. \( \text{FLT} \) admits a \((2\mathbb{H}_n, L - 1 + 1/L)\)-approximate algorithm.

Proof. From the instance of \( \text{FLT} \), create an instance \( I_0 \) of SET COVER as follows. Each facility \( i \) corresponds to a set that covers client \( j \) iff \( d_{ij} \leq B_j/L \). The facilities’ types are ignored. A \( \mathbb{H}_n \)-approximate solution \( S_0 \) is computed for \( I_0 \) (greedy algorithm).

Let \( C_t \) be the clients assigned to a facility of type-\( t \) in \( S_0 \). For every \( t \in [L] \), create an instance \( I_t \) of SET COVER as follows. Each facility \( i \) of type \( t \) corresponds to a set that covers client \( j \) iff \( d_{ij} \leq B_j \). A \( \mathbb{H}_n \)-approximate solution \( S_t \) is computed for \( I_t \). Let \( T \) be an optimal solution to \( \text{FLT} \). Let \( T_0 \) be a subset of \( T \) satisfying \( \forall j \in C, T_0 \) contains at least one facility \( i \in T \) such that \( d_{ij} \leq B_j/L \). Note that \( i \) must exist. As \( T_0 \) is a feasible solution to \( I_0 \), we get that

\[
|S_0| \leq \mathbb{H}_n|T_0| \leq \mathbb{H}_n|T| (2)
\]

For every \( t \in [L] \), let \( T_t \) be the restriction of \( T \) to its facilities of type-\( t \). Since \( T_t \) is a feasible solution to \( I_t \), we get for every \( t \in [L] \) that \( |S_t| \leq \mathbb{H}_n|T_t| \). It follows that

\[
|\bigcup_{t=1}^L S_t| = \sum_{t=1}^L |S_t| \leq \mathbb{H}_n \sum_{t=1}^L |T_t| = \mathbb{H}_n|T| (3)
\]

Combine (2) and (3) to get that \(| \bigcup_{t=0}^L S_t | \leq |S_0| + \big| \bigcup_{t=1}^L S_t \big| \leq 2\mathbb{H}_n|T| \). Since every client \( j \in C_t \) is at distance at most \( B_j/L \) from its assigned facility in \( S_0 \), and at distance at most \( B_j \) from its assigned facility in \( \{I_t : t \in [L] \setminus \{t\} \} \), client \( j \) is at total distance at most \((L - 1 + 1/L)B_j \) from its assigned facilities in \( \bigcup_{t=0}^L S_t \). Thus, \( \bigcup_{t=0}^L S_t \) is \((2\mathbb{H}_n, L - 1 + 1/L)\)-approximate.

2.3 LP-Based Approximations

Consider the \( \text{FLT} \) problem where \( L = 2 \). For each facility \( i \in F \), define a variable \( y_i \) such that \( y_i = 1 \), if the facility \( i \) is open and otherwise 0. For each client \( j \in C \), define \( T_j \) as the subset of \( F_1 \times F_2 \) such that \((i, i') \in T_j \) if and only if \( d_{ij} + d_{i'j} \leq B_j \). Let \( T = \bigcup_{j \in C} T_j \). An LP formulation of our problem is as follows:

\[(\text{LP-A}) \text{ minimize } \sum_{i \in F} y_i \]

subject to:

\[
y_{i} \geq x_{ii'}, \forall (i, i') \in T (4)
\]

\[
\sum_{(i, i') \in T_j} x_{ii'} \geq 1, \forall j \in C (5)
\]

\[
x_{ii'}, y_i \in \{0, 1\} (6)
\]

where \( x_{ii'} = 1 \) means that the pair of facilities \((i, i')\) is opened.

Let \( \phi_j := |T_j| \) and \( \phi := \max_{j \in C} \phi_j \). Since \( \phi \) is the maximum number of facility pairs which can serve a client, it is an adapted notion of frequency. If one solves the relaxation of \( \text{LP-A} \) and open every facility \( i \) such that \( y_i \geq \phi^{-1} \), then the solution is feasible and \( \phi \)-approximate. We are going to define a new parameter \( \psi \) such that \( \psi \leq \phi \) and present an approximation algorithm with performance guarantee \( \psi \).

Fix a client \( j \) and consider the bipartite graph \( G_j \) with vertex set \( V_j \subseteq F \) and edge set \( E_j \). There is an edge \((i, i') \in E_j \) if and only if \( i \in F_1 \), \( i' \in F_2 \), and \( d_{ij} + d_{i'j} \leq B_j \). Equivalently, \((i, i') \in E_j \) if and only if \((i, i') \in T_j \). Furthermore, we impose that every vertex of \( F_j \) must have a positive degree.
Lemma 4. \( S \) is a feasible solution to FLT where \( L = 2 \) if, for every \( j \in C \), there exists a vertex cover \( Q \) of \( G_j \), \( S \cap Q \neq \emptyset \).

Proof. Let \( S \) be a feasible solution. Fix a client \( j \in C \) for which \( S \) contains two facilities \( i_1 \in F_1 \) and \( i_2 \in F_2 \) such that \( d_{ij} + d_{ij} \leq B_j \). In other words, \( G_j \) has an edge \( (i_1,i_2) \). Since every vertex cover \( Q \) of \( G_j \) must contain either \( i_1 \) or \( i_2 \), we have that \( S \cap Q \neq \emptyset \).

Now, let \( S' \) be a subset of \( F \) which intersects every vertex cover \( Q \) of every graph \( G_j \). Suppose by contradiction that \( S' \) is not a feasible solution. At least one client, say \( j' \), is not covered. Thus \( S_{j'} := S' \cap V_{j'} \) is an independent set of \( G_j \). A contradiction is reached because \( V_{j'} \setminus S' \) is a vertex cover of \( G_j \) that \( S' \) does not intersect.

Lemma 4 provides a new formulation of FLT problem inspired from [8]. Let \( Q_j \) denote the set of all vertex covers of \( G_j \), and \( Q := \bigcup_{j \in C} Q_j \).

\[
\begin{align*}
\text{(LP-B)} \quad & \min \sum_{i \in F} y_i \\
\text{subject to:} & \sum_{i \in Q} y_i \geq 1, \forall Q \in Q \\
& y_i \in \{0,1\}, \forall i \in F
\end{align*}
\]

The relaxation of LP-B can be solved in polynomial time (the proof is omitted due to space constraints).

Let \( \tilde{Q}_j \) denote the set of all vertex covers of \( G_j \) that are exclusion-wise minimal, and \( \tilde{Q} := \bigcup_{j \in C} \tilde{Q}_j \). That is, \( \tilde{Q} \) is obtained from \( Q \) by discarding every member \( Q \) such that another member \( \tilde{Q} \) satisfies \( \tilde{Q} \subseteq Q \). Note that a solution to LP-B satisfies \( \forall Q \in \tilde{Q}, \sum_{i \in \tilde{Q}} y_i \geq 1 \), which is (8) where \( Q \) is substituted for \( \tilde{Q} \). Let \( \psi \) denote the size of the largest member of \( \tilde{Q} \). Interestingly, \( \psi \leq \phi \) always holds (the proof is omitted due to space constraints).

Theorem 5. FLT admits a polynomial time \( \psi \)-approximation algorithm when \( L = 2 \).

Proof. Solve the relaxation of LP-B and denote by \( y \) the solution. Guess \( \phi \) (with binary search) and open every facility \( i \) such that \( y_i \geq \psi^{-1} \). The solution is feasible (Lemma 4) and \( \psi \)-approximate. For every \( Q \in \tilde{Q} \), at least one facility \( i \in Q \) satisfies \( y_i \geq |\tilde{Q}|^{-1} \geq \psi^{-1} \). Thus, at least one facility of every \( \tilde{Q} \in \tilde{Q} \) is open, and the same goes for \( Q \).

2.4 Inapproximability

The SYMMETRIC LABEL COVER problem (SLC) is a variant of LABEL COVER introduced in [10] and defined as follows. We are given a complete bipartite graph where \( V_1 \) and \( V_2 \) are the two parts of the bipartition, and \( |V_1| = |V_2| = q \). Two sets of labels \( \mathcal{L}_1 \) and \( \mathcal{L}_2 \) are given. For each \( (v_1,v_2) \in V_1 \times V_2 \), a relation \( R(v_1,v_2) \subseteq \mathcal{L}_1 \times \mathcal{L}_2 \) is given. A feasible solution is a pair of mappings \( \mu_1 : V_1 \rightarrow 2^{\mathcal{L}_1} \) and \( \mu_2 : V_2 \rightarrow 2^{\mathcal{L}_2} \) such that each edge \( (v_1,v_2) \) is consistent, that is there exists a pair \((\ell_1,\ell_2) \in \mu_1(v_1) \times \mu_2(v_2)\) such that \((\ell_1,\ell_2) \in R(v_1,v_2)\). The objective is to minimize \( \sum_{v \in V_1} |\mu_1(v)| + \sum_{v \in V_2} |\mu_2(v)| \). An instance of SLC has size \( \Theta(\sigma) \) where \( \sigma := \sum_{(v_1,v_2) \in V_1 \times V_2} |R(v_1,v_2)| \) [7]. Unless \( \mathbf{NP} \preceq \mathbf{QP} \) (quasi-polynomial time), SLC cannot be approximated within a factor \( O(2^{\log^{\epsilon+\cdot\cdot\cdot}\sigma}) \) for any \( \epsilon > 0 \) [7].

Following the notation of Section 2.3, \( T_j \) is the set of pairs of facilities that cover client \( j \) and let \( \tau := |\bigcup_{j \in C} T_j| \). Thus, the size of an instance of FLT with \( L = 2 \) is \( \Theta(\tau) \).
Theorem 6. Unless $NP \subseteq QP$, FLT with $L = 2$ cannot be approximated within a factor $O(2^{\log^{1/\epsilon} + \tau})$ for any $\epsilon > 0$.

Proof. Take an instance of SLC and build an instance of FLT with $L = 2$ as follows. Each edge $(x, y) \in V_1 \times V_2$ corresponds to a client $j_{xy}$. For each pair $(\ell_1, \ell_2) \in R(x, y)$, for some edge $(x, y)$, create facilities $(x, \ell_1)$ and $(y, \ell_2)$ of types 1 and 2, respectively. We have $(\ell_1, \ell_2) \in R(x, y)$ if the facilities $(x, \ell_1), (y, \ell_2)$ cover $j_{xy}$, i.e. $(x, \ell_1), (y, \ell_2) \in T_{j_{xy}}$.

From a feasible solution to SLC, build a solution with no greater cost: for each edge $(x, y)$, take $\ell_1 \in \mu_1(x)$ and $\ell_2 \in \mu_2(y)$ such that $(\ell_1, \ell_2) \in R(x, y)$, and open facilities $(x, \ell_1)$ and $(y, \ell_2)$. Note that such a pair $(\ell_1, \ell_2)$ exists since $(\mu_1, \mu_2)$ form a feasible solution to the SLC instance. From a feasible solution to FLT, build a solution to SLC having the same cost. At the beginning, $\mu_i(v)$ is empty for every $v \in V_i$ and $i \in \{1, 2\}$. Then, for each open facility $(v, \ell) \in V_i \times L_i$, add $\ell$ to $\mu_i(v)$. In this reduction, $\sigma$ is equal to $\tau$. ▶

### 2.5 Randomized Algorithm

Consider a natural LP formulation of the FLT problem. For each facility $i \in F$, we define a variable $y_i$. For each pair of a client $j$ and a facility $i$, we define a variable $x_{ij}$ such that $x_{ij} = 1$ if $i$ serves $j$. Then the FLT problem can be formulated as:

$$(LP1) \quad \min \sum_{i \in F} y_i$$

subject to

$$y_i \geq x_{ij}, \quad \forall i, j \in F \times C \quad (9)$$

$$\sum_{i \in F} x_{ij} \geq 1, \quad \forall \ell, j \in [L] \times C \quad (10)$$

$$\sum_{i \in F} x_{ij}d_{ij} \leq B_j, \quad \forall j \in C \quad (11)$$

$$x_{ij} \in \{0, 1\}, \quad \forall i, j \in F \times C \quad (12)$$

$$y_i \in \{0, 1\}, \quad \forall i \in F \quad (13)$$

The relaxation of LP1 is obtained by letting the variables $x_{ij}$ and $y_i$ attain fractional values between 0 and 1. Note that the objective value of an optimal solution to the relaxed LP is a lower bound on the objective value of an optimal integral solution. A simple and natural idea for rounding an optimal fractional solution is to consider the fractions as probabilities. Below, we show that this idea leads to a $O(\log n/\epsilon)$-approximation algorithm where the service constraint (11) is relaxed by a factor of at most $(1 + \epsilon)$. The proof of Theorem 7 is omitted due to space constraints.

Theorem 7. There exists a randomized algorithm with the performance guarantee of $O(\log(n)/\epsilon, (1 + \epsilon))$ where $\epsilon \in (0, 1)$ for the FLT problem.

### 3 Metric Instances

In this section, we assume that facilities and clients are placed in a metric space where the distances satisfy the triangle inequality. We next show the following:

Theorem 8. For the FLT problem, there exists a $(1, 3L)$-approximation algorithm when values in the distance matrix $D$ follow triangle inequality.
Algorithm 2: Greedy algorithm in metric space.

Data: $C, \ell \in [L], F_\ell, (B_1, \ldots, B_n)$

1. Initialize $S_\ell \leftarrow \emptyset$, $U \leftarrow C$ and $q \leftarrow 1$

2. while $U \neq \emptyset$ do
   3. Find $j^q \in U$ with smallest budget $B_{j^q}$
   4. Let $p^q_\ell$ be a facility of $F_\ell$ such that $d_{j^q p^q_\ell} \leq B_{j^q}$
   5. $S_\ell \leftarrow S_\ell \cup \{p^q_\ell\}$
   6. Let $C^q_\ell = \{j \in U \mid d_{j p^q_\ell} \leq 3B_{j} \}$
   7. The clients of $C^q_\ell$ are assigned to $p^q_\ell$, and $j^q \in C^q_\ell$ is the representative of $p^q_\ell$
   8. $U \leftarrow U \setminus C^q_\ell$
   9. $q \leftarrow q + 1$

10. return $S_\ell$

Proof. Consider the algorithm 2. The algorithm 2 run for different values of $\ell \in [L]$. For a fix $\ell \in L$, it identifies a set of facilities $S_\ell$ and $|S_\ell|$ representatives. By construction, the representatives $j, j'$ of two different facilities in $S_\ell$ must be at distance strictly larger than $2\max(B_{j}, B_{j'})$ from one another: take the representative $j^q$ of $p^q_\ell$; we have $d_{j^q p^q_\ell} \leq B_{j^q}$. Take a representative $j^q$ of $p^q_\ell$ such that $q > q$. Thus $B_{j^q} \geq B_{j^q}$ and $3B_{j^q} < d_{j^q p^q_\ell}$. By the triangle inequality $d_{j^q p^q_\ell} \leq d_{j^q p^q_\ell} + d_{j^q p^q_\ell}$. We get that $d_{j^q j^q} > 3B_{j^q} - B_{j^q} \geq 2B_{j^q} = 2\max(B_{j}, B_{j'})$.

Because $d_{j^q j^q} > 2\max(B_{j}, B_{j'})$, two representatives $j, j'$ cannot share an $\ell$-facility in the optimum. Therefore there are at least $|S_\ell|$ facilities of type $\ell$ in an optimal solution. It follows that $\bigcup_{\ell=1}^{L} S_\ell$ is a 1-approximation of the optimum concerning the number of open facilities. Since every client $j$ is at distance at most $3B_{j}$ from its assigned facility of type $\ell$, for every $\ell \in [L]$, the approximation ratio on the distance is $3L$. \hfill $\blacksquare$

### 3.1 Euclidean Instances

In this section, we consider instances where the clients and the facilities lie in either $\mathbb{R}$ or $\mathbb{R}^2$ and the Euclidean distance is used. We show that:

- **Proposition 9.** There is an $\mathcal{O}(nm^L(n+m^L))$-time optimal dynamic programming algorithm for linear instances of FLT, where all clients and facilities lie in $\mathbb{R}$.

The proof is omitted due to space constraints. Next, we present the result of Mustafa and Ray [20] for the geometric hitting set problem. The algorithm presented in [20] is a simple local search which starts with any feasible solution (for example open all facilities) and iteratively reduces the size of this set as long as there does not exist a set of $k$ facilities which can be replaced by $k - 1$ facilities, where $k$ is some integer given as an input. This algorithm is known as a $k$-level local search algorithm. Their main result is the following:

- **Lemma 10.** There exists a constant $c$ such that $(c/\epsilon)^2$-level local search algorithm returns a hitting set of size at most $(1 + \epsilon)$ times the size of an optimal hitting set where $\epsilon \in (0, 1)$ [20].

If the same reasoning as for propositions 3 (replace the greedy algorithm with the PTAS of [20]), then in $\mathbb{R}^2$, FLT admits approximation algorithms with guarantees $(1 + \epsilon, L)$ and $(2 + \epsilon, L - 1 + 1/L)$. 

ISAAC 2018
3.1.1 A Local Search Algorithm in $\mathbb{R}^2$

Recall that $L$ is a constant. We say that $S$ is $\epsilon$-feasible if each client $j$ is served by a type-$\ell$ facility and the total service cost for $j$ is at most $(1+\mathcal{O}(\epsilon))B_j$ for $0 < \epsilon < 1$. Let $S$ and $S'$ denote two $\epsilon$-feasible solutions. Then $S \oplus S'$ denotes the symmetric difference between $S$ and $S'$, that is $S \oplus S' := (S' - S) \cup (S - S')$.

**Local Search Algorithm.** Start with any $\epsilon$-feasible solution $S$. While possible, replace $S$ with an $\epsilon$-feasible set of facilities $S'$ such that $|S'| < |S|$ and $|S \oplus S'| \leq \mathcal{O}(1/\epsilon^4)$.

Observe that the local search algorithm is similar to the $k$-level local search algorithm mentioned in [20]. The only difference is in the definition of feasibility. That is, a solution in the $k$-level local search algorithm is considered feasible if the budget for each client $j$ is at most $B_j$ (the budget corresponds to the radius of disks), whereas our local search algorithm relaxes the budget for each client by a factor of $1 + \mathcal{O}(\epsilon)$.

**Lemma 11.** The running time of the local search algorithm is polynomial in the size of the input.

**Proof.** An initial $\epsilon$-feasible solution is to open, for each client $j$, the closest facility of each type $\ell \in [L]$. Hence the initial solution opens at most $nL$ facilities. Since in each iteration the local search algorithm reduces the number of facilities by at least one, the total number of iterations is at most $nL$. In each iteration, the number of possible different combinations to check is at most $(\mathcal{O}(1/\epsilon^4))$. Hence the total running time of the algorithm is $nLm^{\mathcal{O}(1/\epsilon^4)}$. The lemma follows since $L$ is a constant.

**Theorem 12.** Assume that clients have uniform upper bound on the service cost, that is, $\forall j \in C, B_j = B$. Then, the local search algorithm achieves a $(1+\mathcal{O}(\epsilon), 1+\mathcal{O}(\epsilon))$-approximation ratio for the FLT problem in $\mathbb{R}^2$ where $\epsilon \in (0, 1)$.

**Proof.** We assume w.l.o.g. that $\frac{1}{\epsilon}$ is an integer, and that the set of clients $C$ and the set of facilities $F$ are enclosed in an area $A$. Let $R$, $R^{2B}$ and $R^{4B}$ denote squares centered at a given point $p$ of width $\frac{2B}{\epsilon}$, $\left(\frac{2B}{\epsilon} + 2B\right)$, and $\left(\frac{2B}{\epsilon} + 4B\right)$, respectively. Let $ALG$ and $OPT$ denote the set of facilities opened by the local search algorithm and in some fixed optimal solution, respectively. Further, let $ALG(R')$ and $OPT(R')$ represent the restrictions of $ALG$ and $OPT$ to the square $R'$, respectively.

Next, we grid the entire region $A$ such that the internode distance is $\epsilon B$. Let $K$ denote the set of small squares of width $\epsilon B$. Let $OPT'$ be a solution such that for each tiny square $k \in K$ and each type-$\ell \in [L]$, one type-$\ell$ facility is opened if and only if $OPT$ has at least one open type-$\ell$ facility in $k$. Thus, we have $|OPT'| \leq |OPT|$. Let $OPT'(R')$ represent the set of facilities open inside the region $R'$ in $OPT'$. Also, we have $|OPT'(R^{4B})| \leq |OPT(R^{4B})|$

Consider the intermediate solution $M$ formed by removing all the facilities opened by the local search algorithm in the square $R$ from $ALG$ and adding all the facilities opened in $OPT'$ inside the square $R^{4B}$ that is, $M = (ALG \setminus ALG(R)) \cup OPT'(R^{4B})$

**Claim 13.** $M$ forms an $\epsilon$-feasible solution to the FLT problem.

**Proof.** Observe that the facilities opened inside the square $R$ can serve clients in the square $R^{2B}$. Therefore, closing the facilities inside $R$ can lead to infeasible solution for clients in the square $R^{2B}$. Let $j$ be some client enclosed in square $R^{2B}$. Let $F_j'$ denote the set of facilities
We introduce and study the approximability of covering problems with multiple types and a
where the clients arrive one-by-one and must be covered by a facility of each type upon
Recall that \( OPT' \), for each tiny square with width \( \epsilon B \) and each type \( \ell \), opens a facility of
type-\( \ell \) if and only if \( OPT' \) has at least an open facility type-\( \ell \) in the same square. Thus for
each type \( \ell \) facility in \( F_j' \), there exists a facility a type \( \ell \) facility in \( OPT'(R_{4\epsilon B}) \) such that
the service cost of \( j \) to a type-\( \ell \) facility is at most the service cost of \( j \) to type-\( \ell \) in \( OPT \) plus
\( \sqrt{2\epsilon B} \). Summing over all types, the claim follows. ▶

Observe that \( |M \oplus ALG| \) can be much larger than \( O(\frac{1}{\epsilon^2}) \) if \( ALG(R) \) is huge. However, \( M \)
can be realized after few steps of the local search algorithm wherein each iteration, the
local search algorithm closes \( O(\frac{1}{\epsilon^2}) \) facilities from \( ALG(R) \setminus OPT'(R_{4\epsilon B}) \). Thus, the local
exchange argument states that
\[
|ALG| \leq |M| = |(ALG \setminus ALG(R)) \cup OPT'(R_{4\epsilon B})| \\
|ALG \setminus ALG(R)| + |ALG(R)| \leq |(ALG \setminus ALG(R))| + |OPT'(R_{4\epsilon B})| \\
|ALG(R)| \leq |OPT'(R)| + |OPT'(R_{4\epsilon B} - R)|
\]
Let \( R^P \) denote the set of all regions in \( A \) according to some partitioning scheme \( P \). For each
region \( R \in R^P \), the above local exchange argument holds. Summing over all regions, we have
\[
\sum_{R \in R^P} |ALG(R)| \leq \sum_{R \in R^P} (|OPT'(R)| + |OPT'(R_{4\epsilon B} - R)|) \\
\leq |OPT'| + \sum_{R \in R^P} |OPT'(R_{4\epsilon B} - R)|
\]
▶ Claim 14. There exists a partition \( Q \) such that \( \sum_{R \in R^Q} |OPT'(R_{4\epsilon B} - R)| = O(\epsilon)|OPT'|. \)
Proof. In this proof, we use the idea of the “grid shifting strategy” mentioned in [16]. Due to
space constraints, the proof is omitted. ▶

From above claim it follows that \( |ALG| \leq (1 + O(\epsilon))|OPT'| \leq (1 + O(\epsilon))|OPT|. \) ▶

4 Conclusion and Directions for Further Research

We introduce and study the approximability of covering problems with multiple types and a
hard constraint on the combined “quality” of each client’s covering. Our work leaves many
promising directions for future research. A natural question is whether we could obtain
strong approximation guarantees for metric instances of constant doubling dimension.

Given that it is very difficult, if even possible, to achieve good approximation guarantees
without violating the budget constraint (for both general and metric instances), it would be
very interesting to investigate the approximability of \( \text{FLT} \) with penalties (see e.g., [19, 5] for
the approximability of other covering problems with penalties). In \( \text{FLT} \) with penalties, there
is a covering penalty, which is a non-decreasing function of each client’s total covering cost.
In the simplest case, the covering penalty is 0, if the budget constraint is satisfied, and some
\( p_j > 0 \), if the budget constraint is not satisfied for client \( j \). The cost of the solution is the
sum of the facility opening costs and the covering penalties for all clients.

Another natural research direction is to determine the competitive ratio of online \( \text{FLT} \),
where the clients arrive one-by-one and must be covered by a facility of each type upon
arrival. A promising starting point is the ideas and techniques applied to \( \text{ONLINE SET COVER} \)
[2] for general instances and to \( \text{ONLINE FACILITY LOCATION} \) problems (see e.g., [13] and the
references therein) for metric instances.
Covering Clients with Types and Budgets

References


