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 (Reykjavik University, IS and Gran Sasso Science Institute, IT)
- Christel Baier (TU Dresden, DE)
- Roberto Di Cosmo (Inria and Université Paris Cité, FR)
- Faith Ellen (University of Toronto, CA)
- Javier Esparza (TU München, DE)
- Daniel Král’ (Masaryk University, Brno, CZ)
- Meena Mahajan (Chair, Institute of Mathematical Sciences, Chennai, IN)
- Anca Muscholl (University of Bordeaux, FR)
- Chih-Hao Luke Ong (Nanyang Technological University, SG)
- Phillip Rogaway (University of California, Davis, US)
- Eva Rotenberg (Technical University of Denmark, Lyngby, DK)
- Raimund Seidel (Universität des Saarlandes, Saarbrücken, DE and Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Wadern, DE)
- Pierre Senellart (ENS, Université PSL, Paris, FR)

ISSN 1868-8969

https://www.dagstuhl.de/lipics
Dedicated to the memory of Nick Higham.
## Contents

**Preface**  
*Leo Liberti* ✂️ 0:ix

**Permanent SEA committees** ✂️ 0:xi

**Organization of SEA 2024** ✂️ 0:xiii–0:xiv

**Authors** ✂️ 0:xv–0:xviii

### Regular Papers

**Move-r: Optimizing the r-index**  
*Nico Bertram, Johannes Fischer, and Lukas Nalbach* ✂️ 1:1–1:19

**Engineering Zuffix Arrays**  
*Paolo Boldi, Stefano Marchini, and Sebastiano Vigna* ✂️ 2:1–2:18

**Practical Minimum Path Cover**  
*Manuel Cáceres, Brendan Maney, Santeri Toivonen, and Alexandru I. Tomescu* ✂️ 3:1–3:19

**Separator Based Data Reduction for the Maximum Cut Problem**  
*Jonas Charfreitag, Christine Dahn, Michael Kaibel, Philip Mayer, Petra Mutzel, and Lukas Schürmann* ✂️ 4:1–4:21

**Buffered Streaming Edge Partitioning**  
*Adil Chhabra, Marcelo Fonseca Faraj, Christian Schulz, and Daniel Seemaier* ✂️ 5:1–5:21

**Faster Treewidth-Based Approximations for Wiener Index**  
*Giovanna Kobus Conrado, Amir Kafshdar Goharshady, Pavel Hudec, Pingjiang Li, and Harshit Jitendra Motwani* ✂️ 6:1–6:19

**Local Search k-means++ with Foresight**  
*Theo Conrads, Lukas Drexler, Joshua Könen, Daniel R. Schmidt, and Melanie Schmidt* ✂️ 7:1–7:20

**Practical Computation of Graph VC-Dimension**  
*David Coudert, Mónika Csikós, Guillaume Ducoffe, and Laurent Viennot* ✂️ 8:1–8:20

**Top-k Frequent Patterns in Streams and Parameterized-Space LZ Compression**  
*Patrick Dinklage, Johannes Fischer, and Nicola Prezza* ✂️ 9:1–9:20

**Taxonomic Classification with Maximal Exact Matches in KATKA Kernels and Minimizer Digests**  
*Dominika Draesslerová, Omar Ahmed, Travis Gagie, Jan Holub, Ben Langmead, Giovanni Manzini, and Gonzalo Navarro* ✂️ 10:1–10:13

**Engineering Weighted Connectivity Augmentation Algorithms**  
Streaming Matching and Edge Cover in Practice
S M Ferdous, Alex Pothen, and Mahantesh Halappanavar ........................ 12:1–12:22

3/2-Dual Approximation for CPU/GPU Scheduling

Accelerating ILP Solvers for Minimum Flow Decompositions Through Search
Space and Dimensionality Reductions
Andreas Grigorjew, Fernando H. C. Dias, Andrea Cracco, Romeo Rizzi, and
Alexandru I. Tomescu ........................................................... 14:1–14:19

Determining Fixed-Length Paths in Directed and Undirected Edge-Weighted
Graphs
Daniel Hambly, Rhyd Lewis, and Padraig Corcoran .............................. 15:1–15:11

Solving the Optimal Experiment Design Problem with Mixed-Integer Convex
Methods
Deborah Hendrych, Mathieu Besançon, and Sebastian Pokutta ................... 16:1–16:22

Barcode Selection and Layout Optimization in Spatial Transcriptomics
Frederik L. Jatzkowski, Antonia Schmidt, Robert Mank, Steffen Schüler, and
Matthias Müller-Hannemann .................................................... 17:1–17:19

Experimental Analysis of LP Scaling Methods Based on Circuit Imbalance
Minimization
Jakub Komárek and Martin Koutecký ........................................... 18:1–18:21

Algorithms for Gradual Polyline Simplification
Nick Krumbholz, Stefan Funke, Peter Schäfer, and Sabine Storandt .............. 19:1–19:18

Targeted Branching for the Maximum Independent Set Problem Using Graph
Neural Networks
Kenneth Langedal, Demian Hespe, and Peter Sanders ............................ 20:1–20:21

SPIDER: Improved Succinct Rank and Select Performance
Matthew D. Laws, Jocelyn Bliven, Kit Conklin, Elyes Laalai, Samuel McCauley,
and Zach S. Sturdevant .......................................................... 21:1–21:18

Efficient Computation of Topological Integral Transforms

Engineering A* Search for the Flip Distance of Plane Triangulations

Efficient Exact Online String Matching Through Linked Weak Factors
Matthew N. Palmer, Simone Faro, and Stefano Scaflti ............................ 24:1–24:14

Convex Relaxation for the Generalized Maximum-Entropy Sampling Problem
Gabriel Ponte, Marcia Fampa, and Jon Lee ...................................... 25:1–25:14

Scalable Hard Instances for Independent Set Reconfiguration
Takehide Soh, Takuma Watanabe, Jun Kawahara, Akira Suzuki, and Takehiro Ito 26:1–26:15

Improved Cut Strategy for Tensor Network Contraction Orders
Christoph Staudt, Mark Blacher, Julien Klaus, Färin Lippmann, and
Joachim Giesen ................................................................. 27:1–27:19

Finding the Minimum Cost Acceptable Element in a Sorted Matrix
Sebastián Urrutia and Vinicius dos Santos ...................................... 28:1–28:14
Preface

This volume contains the papers accepted for presentation at the 22nd edition of the International Symposium on Experimental Algorithms (SEA 2024), held in Vienna from 23rd to 26th July 2024.

SEA, previously known as the Workshop of Experimental Algorithms (WEA), is an international forum for researchers in the area of design, analysis, engineering, and experimental evaluation algorithms, as well as in various aspects of computational optimization and applications. The symposium aims at attracting papers from both the Computer Science and the Operations Research & Mathematical Programming communities. Submissions to SEA are required to present significant contributions supported by experimental evaluation, methodological advances in the design and interpretation of experiments, scientifically grounded use cases of heuristics and meta-heuristics, application-driven case studies that deepen the understanding of a given problem. The main goal of SEA is the creation of a friendly environment leading to the establishment or strengthening of scientific collaborations and exchanges among participants.

Starting this year, the SEA Steering Committee (SC) updated the list of “topics of interest” of the symposium to include cryptography and security, databases, and natural language processing. No-one in the SC expected this change to immediately yield results, obviously: and, indeed, this year’s scientific program hardly reflects the update, aside from a couple of submissions that could not be accepted for presentation. But I think it is important to emphasize this change as something forward-looking. Another change was proposed by SC members as the organization of SEA24 was already underway: a double-blind reviewing process. Although I am favorable to this change, I proposed that it should be made effective in a next edition of SEA, since I had already started the PC activities with the classic model. It is likely that this change will be carried out already with SEA25.

Each submission to SEA 2024 was reviewed by at least three Program Committee (PC) members or their chosen reviewers. After a careful peer review and evaluation process, 28 papers out of 60 submissions were accepted for presentation at the symposium and inclusion in this LIPIcs proceedings volume. The selection rate was 46.7%. The scientific program of the symposium originally contained three plenary lectures: Nick Higham, Pascal Schweitzer, and Yllka Velaj. Alas, one of these speakers was lost to a fatal illness: this volume is therefore dedicated to Nick’s memory. The selection of the best paper was carried out by calling all PC members to vote on the five accepted papers with highest ranking from the selection process (any PC member who was a co-author of these five highest-ranking paper was barred from voting). The outcome of this vote will be announced at the conference.

The local organizing committee of this twenty-second edition of SEA was chaired by Kathrin Hanauer of the Theory and Application of Algorithms research group at Vienna University. Special thanks go to the members of the organizing committee: Sophia Heck, Lara Ost, and Maximilian Vötsch. Thanks are also due to Universität Wien and the Fakultät für Informatik for their support with organizing and hosting the conference. All of the SEA committees are extremely grateful to our sponsors, the Vienna Convention Bureau and Google Research, who helped SEA cover part of the expenses and keep student registration fees affordable.

In the last several years, SEA symposia have always invited the authors of a selection of the accepted papers to submit an extended version of their work to a special issue of the ACM Journal of Experimental Algorithmics dedicated to SEA. This journal has now ceased
publication, so the SC strategy for this edition of SEA is tentative. On the one hand, the authors of a few selected papers accepted at SEA 2024 shall receive an invitation to submit an extended version of their contribution to the journal *Algorithmica*. On the other hand, if there is sufficient interest, I will look for a dedicated post-proceedings special issue in another journal.

I wish to express my gratitude to a number of people who have helped me considerably in my PC Chair role: first and foremost, all of the members of the PC, and their chosen sub-reviewers. Secondly, the chair of the local organizing committee Kathrin Hanauer for a seamless integration of the two main committees for SEA 2024 (PC and local). Third, some SC members who had previously organized some editions of SEA, and who gave me extremely valuable advice. Lastly, I am grateful to the SC Chair, Ulrich Meyer, for answering numerous questions timely, precisely, helpfully, and with an extremely supporting and trusting attitude.

As a final comment, I can now confess to having read Ulrich’s initial email very fast (this email bore the invitation to chair the PC of SEA 2024). I read it so fast, in fact, that I had understood it as an invitation to *join* rather than *chair* the PC. I therefore quickly answered “count me in” and got back to work. Ulrich’s subsequent answer exuded an amount of relief and enthusiasm that sounded suspiciously excessive. Only then did I go back to his invitation, and realized what I had gotten myself into. At the end of the process, I must say that this was a serendipitous misunderstanding: I was very happy to chair this committee, and to help SEA 2024 achieve an interesting scientific program.

Leo Liberti
Palaiseau, France
Permanent SEA committees

Advisory Board

- Josep Díaz, Universitat Politècnica de Catalunya, Spain
- Giuseppe Italiano, LUISS University, Italy
- Klaus Jansen, University of Kiel, Germany
- Ian Munro, University of Waterloo, Canada
- Sotiris Nikoletseas, Patras University, Greece
- Jose Rolim, University of Geneva, Switzerland
- Pavlos Spirakis, University of Liverpool, UK

Steering Committee

- Gianlorenzo D’Angelo, Gran Sasso Science Institute, Italy
- Domenico Cantone, Università degli Studi di Catania, Italy
- David Coudert, INRIA, France
- Simone Faro, Università di Catania, Italy
- Loukas Georgiadis, University of Ioannina, Greece
- Leo Liberti, CNRS and École Polytechnique, France
- Ulrich Meyer, Goethe University Frankfurt, Germany (chair)
- Emanuele Natale, CNRS, Université Cote d’Azur, I3S, INRIA, France
- Gonzalo Navarro, University of Chile, Chile
- Cynthia Phillips, Sandia National Laboratories, USA
- Simon Puglisi, University of Helsinki, Finland
- Christian Schulz, Heidelberg University, Germany
- Sabine Storandt, University of Konstanz, Germany
- Laurent Viennot, INRIA, France
- Bora Uçar, CNRS, Laboratoire LIP, Lyon, France
- Dorothea Wagner, Karlsruhe Institute of Technology, Germany
- Christos Zaroliagis, University of Patras, Greece
Organization of SEA 2024

Local organizing committee

- Kathrin Hanauer (chair)
- Sophia Heck
- Lara Ost
- Maximilian Vötsch

The members of the organizing committee are from the research group “Theory and Application of Algorithms” at the Faculty of Computer Science at the University of Vienna.

Program committee

- Pietro Belotti, Politecnico di Milano, Italy
- Ksenia Bestuzheva, ZIB, Germany
- Davide Buscaldi, Université Sorbonne Paris Nord, France
- Martina Cerulli, Università di Salerno, Italy
- Roberto Cordone, Università di Milano, Italy
- David Coudert, INRIA, France
- Silvia Di Gregorio, Université Sorbonne Paris Nord, France
- Simone Faro, Università di Catania, Italy
- Irene Finocchi, LUISS University, Italy
- Donatella Firmani, Università di Roma La Sapienza, Italy
- Kathrin Hanauer, University of Vienna, Austria
- Claudio Gentile, IASI-CNR Rome, Italy
- Loukas Georgiadis, University of Ioannina, Greece
- Daniel Gonçalves, CNRS and Université de Montpellier, France
- Douglas Gonçalves, Federal University of Santa Catarina, Brazil
- Marc Goerigk, University of Passau, Germany
- Ralf Klasing, CNRS and Université de Bordeaux, France
- Susana Ladrera, Universidade da Coruña, Spain
- Leo Liberti, CNRS and École Polytechnique, France (chair)
- Vicky Papadopoulou-Lesta, European University, Cyprus
- Georgios Paschos, Amazon Science, Luxembourg
- Cynthia A. Phillips, Sandia National Laboratories, USA
- Mustafa Pinar, Bilkent University, Turkey
- Pierre-Louis Poirion, RIKEN Institute Tokyo, Japan
- Leena Salmela, University of Helsinki, Finland
- Melanie Schmidt, HHU Düsseldorf, Germany
- Christian Schulz, University of Heidelberg, Germany
- Andrea Scozzari, Niccolò Cusano University Rome, Italy
- Maria José Serna Iglesias, Universitat Politècnica de Catalunya, Spain
- Sabine Storandt, University of Konstanz, Germany
- Bora Uçar, CNRS and ENS Lyon, France
- Domagoj Vrgoč, Pontificia Universidad Católica, Chile
- Annegret Wagler, Université de Clermont Auvergne, France
External Reviewers

List of Authors

Omar Ahmed (10)
Johns Hopkins University, Baltimore, MD, USA

Nico Bertram (1)
Technische Universität Dortmund, Germany

Mathieu Besançon (16)
Université Grenoble Alpes, Inria, LIG, Grenoble, France; Zuse Institute Berlin, Germany

Mark Blacher (27)
Friedrich Schiller University Jena, Germany

Jocelyn Bliven (21)
Williams College Computer Science, Williamstown, MA, USA

Paolo Boldi (2)
Università degli Studi di Milano, Italy

Jonas Charfreitag (4)
Institute of Computer Science, University of Bonn, Germany

Adil Chhabra (5)
Heidelberg University, Germany

Kit Conklin (21)
Williams College Computer Science, Williamstown, MA, USA

Giovanna Kobus Conrado (6)
Hong Kong University of Science and Technology (HKUST), Clear Water Bay, New Territories, Hong Kong

Theo Conrads (7)
Department of Computer Science, University of Cologne, Germany

Padraig Corcoran (15)
School of Computer Science and Informatics, Cardiff University, Wales, UK

David Coudert (8)
Université Côte d’Azur, Inria, I3S, CNRS, France

Andrea Cracco (14)
University of Verona, Italy

Mónika Csikós (8)
IRIF, CNRS and Université Paris Cité, Paris, France

Manuel Cáceres (3)
Department of Computer Science, University of Helsinki, Finland; Department of Computer Science, Aalto University, Finland

Christine Dahn (4)
Institute of Computer Science, University of Bonn, Germany

Fernando H. C. Dias (14)
Aalto University, Espoo, Finland

Patrick Dinklage (9)
TU Dortmund University, Germany

Vinicius dos Santos (28)
Federal University of Minas Gerais, Brazil

Dominika Draesslerová (10)
Czech Technical University in Prague, Czech Republic

Lukas Drexler (7)
Faculty of Mathematics and Natural Sciences, Department of Computer Science, Heinrich Heine University Düsseldorf, Germany

Guillaume Ducoffe (8)
University of Bucharest, Romania; National Institute for Research and Development in Informatics, Bucharest, Romania

Marcia Fampa (25)
Universidade Federal do Rio de Janeiro, Brazil

Marcelo Fonseca Faraj (11)
Heidelberg University, Heidelberg, Germany

Simone Faro (24)
Department of Mathematics and Computer Science, University of Catania, Italy

S M Ferdous (12)
Pacific Northwest National Laboratory, Richland, WA, USA

Johannes Fischer (1, 9)
Technische Universität Dortmund, Germany

Marcelo Fonseca Faraj (5)
Heidelberg University, Germany

Stefan Funke (19)
University of Stuttgart, Germany

Travis Gagie (10)
CeBiB & Dalhousie University, Halifax, Canada
Stefano Marchini (2)
Università degli Studi di Milano, Italy

Philip Mayer (4, 23)
Institute of Computer Science, University of Bonn, Germany

Samuel McCauley (21)
Williams College Computer Science, Williamstown, MA, USA

Harshit Jitendra Motwani (6)
Department of Computer Science and Engineering & Department of Mathematics, Hong Kong University of Science and Technology (HKUST), Clear Water Bay, New Territories, Hong Kong

Brendan Mumey (3)
School of Computer Science, Montana State University, Bozeman, MT, USA

Petra Mutzel (4, 23)
Institute of Computer Science, University of Bonn, Germany

Thomas Möller (11)
Heidelberg University, Heidelberg, Germany

Matthias Müller-Hannemann (17)
Martin Luther University Halle-Wittenberg, Germany

Lukas Nalbach (1)
Technische Universität Dortmund, Germany

Gonzalo Navarro (10)
CeBiB & DCC, University of Chile, Chile

Felix Ohnesorge (13)
Kiel University, Germany

Steve Oudot (22)
Inria Saclay, Palaiseau, France

Matthew N. Palmer (24)
The British Computer Society, Swindon, United Kingdom

Hugo Passe (22)
Ecole Normale Supérieure de Lyon, France

Sebastian Pokutta (16)
Technische Universität Berlin, Germany; Zuse Institute Berlin, Germany

Gabriel Ponte (25)
University of Michigan, Ann Arbor, MI, USA

Alex Pothen (12)
Purdue University, West Lafayette, IN, USA

Nicola Prezza (9)
Ca’ Foscari University of Venice, Italy

Romeo Rizzi (14)
University of Verona, Italy

Peter Sanders (20)
Karlsruhe Institute of Technology, Institute for Theoretical Informatics, Germany

Stefano Scafiti (24)
Department of Mathematics and Computer Science, University of Catania, Italy

Antonia Schmidt (17)
Martin Luther University Halle-Wittenberg, Germany

Daniel R. Schmidt (7)
Faculty of Mathematics and Natural Sciences, Department of Computer Science, Heinrich Heine University Düsseldorf, Germany

Melanie Schmidt (17)
Faculty of Mathematics and Natural Sciences, Department of Computer Science, Heinrich Heine University Düsseldorf, Germany

Christian Schulz (5, 11)
Heidelberg University, Germany

Peter Schäfer (19)
University of Konstanz, Germany

Steffen Schüler (17)
Martin Luther University Halle-Wittenberg, Germany

Lukas Schürmann (4)
Institute of Computer Science, University of Bonn, Germany

Takehide Soh (26)
Information Infrastructure and Digital Transformation Initiatives Headquarters, Kobe University, Japan

Christoph Staudt (27)
Friedrich Schiller University Jena, Germany

Sabine Storandt (19)
University of Konstanz, Germany

Zach S. Sturdevant (21)
Williams College Computer Science, Williamstown, MA, USA
Authors

Akira Suzuki (26)
Graduate School of Information Sciences,
Tohoku University, Japan

Santeri Toivonen (3)
Department of Computer Science, University of
Helsinki, Finland

Alexandru I. Tomescu (3, 14)
Department of Computer Science, University of
Helsinki, Finland

Malte Tutas (13)
Kiel University, Germany

Sebastián Urrutia (28)
Molde University College, Norway

Laurent Viennot (8)
Inria, DI ENS, Paris, France

Sebastiano Vigna (2)
Università degli Studi di Milano, Italy

Takumu Watanabe (26)
Graduate School of Information Sciences,
Tohoku University, Japan
Move-r: Optimizing the r-index

Nico Bertram
Technische Universität Dortmund, Germany

Johannes Fischer
Technische Universität Dortmund, Germany

Lukas Nalbach
Technische Universität Dortmund, Germany

Abstract

We present a static text index called Move-r, which is a highly optimized version of the r-index ([11] Gagie et al., 2020) that incorporates recent theoretical developments of the move data structure ([19] Nishimoto and Tabei, 2021). The r-index is the method of choice for indexing highly repetitive texts, such as different versions of a text document or DNA from the same species, as it exploits the compressibility of the underlying data. With Move-r, we can answer count- and locate queries 2–35 (typically 15) times as fast as with any other r-index supporting locate queries while being 0.8–2.5 (typically 2) times as large. A Move-r index can be constructed 0.9–2 (typically 2) times as fast while using 1/3–1 (typically 1/2) times as much space.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Compressed Text Index, Burrows-Wheeler Transform

Introduction

Answering pattern matching queries on repetitive texts is a common task in bioinformatics, in particular when indexing DNA (assembled or unassembled) from the same species. In such situations, it is important to exploit the repetitiveness of the data, and not use indexes that store the whole data uncompressed.

The r-index [11] is a recent and important development in this area, and uses $O(r)$ words of space, where $r$ is the number of equal-letter runs in the Burrows-Wheeler Transformation, which is an accepted measure for compressibility of highly similar texts [17]. The r-index answers counting queries (counting the number of matches of a length-$m$ pattern) in $O(m \log \log (\sigma + n/r))$ time, and locate queries (listing all matching positions) in additional $O(\text{occ} \log \log (n/r))$ time, where $n$ is the total text length, $\text{occ}$ is the number of matches, $\sigma$ is the alphabet size, and $w$ the word width of the word-RAM.

The bottleneck during the computation of those queries are predecessor queries. Recently, the so-called move data structure has been proposed [19], which can answer the r-index-specific predecessor queries in $O(1)$ time. The move data structure uses the property that two important functions (LF and $\Phi$, see next section for their definitions) can be divided into $r$ intervals where the function values increase by exactly 1. Due to specific access patterns of those functions during a pattern query, it is possible to subdivide those intervals (a process called balancing) such that the functions can be evaluated in $O(1)$ time, while still using only $O(r)$ space.
Our Contributions. The practical performance of the move data structure is still largely open: how can it be constructed efficiently, how fast does it answer queries, and how does it compare with other implementations of the r-index? Does it even lead to a practical advantage? We answer these questions and make the following contributions with a data structure that we call Move-r:

- We present a fast construction algorithm for the move data structure (Section 4.1). While the general idea of the move data structure is simple to implement, it is not obvious how to practically perform the balancing step (Section 3.2) efficiently.
- We present a practically optimized locate-algorithm (Section 4.2) that significantly reduces the number of memory accesses, at the cost of an asymptotically worse running time.
- We introduce additional practical improvements to the move data structure that reduce the space and improve cache efficiency (Appendix B).
- From our experiments we conclude that the theoretical benefit of the $O(1)$ time over the $O(\log \log(n/r))$ time to answer predecessor queries is reflected in practice, since although Move-r indexes are $0.8–2.5$ (typically $2$) times as large as the fastest other r-index supporting locate queries, they can answer count- and locate queries $2–35$ (typically $15$) times as fast. Move-r can also be constructed $0.9–2$ (typically $2$) times as fast (and $2.5–30$, typically $20$, times as fast as any dynamic r-index), while leaving a memory footprint that is $1/3–1$ (typically $1/2$) times as large as those of other static r-indexes and sometimes even competitive with dynamic r-indexes.

2 Preliminaries

In the following, we define necessary notations for the rest of this paper. An interval $[i, j]$ describes the set $\{i, ..., j\}$. For convenience, we use the following notations: $[i, j] = [i, j + 1) = (i - 1, j] = (i - 1, j + 1)$. Given a set $S \subseteq U$, we define the predecessor of $i \in U$ in $S$ as $\text{pred}(i) = \max\{j | j \in S \land j < i\}$. Similarly, we define the successor of $i \in U$ in $S$ as $\text{succ}(i) = \min\{j | j \in S \land j > i\}$. We assume that all described algorithms work in the word-RAM model with word size $w$ [13].

An alphabet $\Sigma$ is a finite ordered set of symbols of size $|\Sigma| = \sigma$. A string $T \in \Sigma^*$ with $|T| = n$ is a sequence of symbols in the alphabet $\Sigma$. In case that $|T| = 0$, $T$ is the empty string $\epsilon$. We can access the $i$-th symbol of $T$ with $T[i]$. A substring $T[i, j]$ is defined by $T[i] \cdot T[i + 1] \cdot ... \cdot T[j]$. In case that $i > j$, we set $T[i, j] = \epsilon$. The substring $T[1, i]$ is called the $i$-th prefix of $T$ and the substring $T[i, n]$ is called the $i$-th suffix of $T$ and is denoted by $T_i$. In case that the length of a prefix or suffix is at least 1 and at most $n - 1$ we call them proper. We assume an order $c_1 < c_2 < ... < c_\sigma$ on the alphabet $\Sigma = \{c_1, c_2, ..., c_\sigma\}$. The lexicographic order of strings is then defined by $T < T' \iff T$ is a proper prefix of $T' \lor \exists i: T[1, i] = T'[1, i] \land T[i + 1] < T'[i + 1]$. We are also interested in how often a character $c$ appears in a string $T[1, i]$, which we denote by $\text{rank}(T, c, i)$.

2.1 Burrows-Wheeler Transform

In the following, each string $T \in \Sigma^*$ is terminated by a sentinel symbol $\$ that is lexicographically smaller than all other symbols in $\Sigma$. This allows us to simplify our algorithms.

The suffix array [14] $SA$ of a string $T$ consists of the starting positions of all suffixes of $T$ in their lexicographic order, i.e., $T_{SA}[1] < T_{SA}[2] < ... < T_{SA}[n]$. For a pattern $P \in \Sigma^*$, there is a maximum interval in the suffix array (the suffix array interval of $P$) containing exactly the positions of all occurrences of $P$ in $T$. For a string $T$, the $i$-th rotation is defined by $T[i, n]T[1, i - 1]$. The rotation matrix is then the $n \times n$ matrix where the $i$-th row consists of
the \(i\)-th rotation of \(T\). When we sort the rotation matrix lexicographically, the last column \(L\) is then called the Burrows-Wheeler Transform (BWT) [6] of \(T\) and the first column is denoted by \(F\). The BWT has the property that same symbols are often grouped runs of the same symbols. For that reason, we can compress the BWT by applying a run-length encoding to it. Let \(L_1, L_2, ..., L_r\) denote those runs, i.e., \(r\) is the number of runs in \(L\) and \(L = L_1 \cdot L_2 \cdot ... \cdot L_r\). Then we can compress the BWT into the run-length encoded BWT (RLBWT) \((L_1[1], |L_1|), (L_2[1], |L_2|), ..., (L_r[1], |L_r|)\).

The last-to-first (LF) mapping of a string maps each symbol in the BWT to the corresponding symbol in \(F\) and is defined by \(LF(i) = j \Leftrightarrow SA[j] = SA[i] - 1\) if \(SA[i] > 1\), and \(LF(i) = 1\) if \(SA[i] = 1\). The LF property describes that the order of same symbols is identical in \(F\) and \(L\), i.e., \(L[i] = L[j] \land i < j \Rightarrow LF(i) < LF(j)\). Given the BWT \(L\) of a string \(T\), we can augment \(L\) with a rank data structure and the frequency array \(C\) of each character with \(C[c] = |\{i \in [1, n] | T[i] = c\}\). Then, \(LF(i) = C[L[i]] + \text{rank}(L, L[i], i)\) because equal symbols in \(F\) are grouped together, and the LF property. Thus, the following important property holds, which allows us to implement LF in \(O(r)\) space:

\[\text{Lemma 1. Let } T \text{ be a string of length } n \text{ and } L \text{ be its BWT. Let } l_1 < l_2 < ... < l_r \text{ be the starting positions of each run in } L \text{ and let } l_{r+1} = n + 1. \text{ Then } LF(i) = LF(l_x) + (i - l_x) \text{ holds for } l_x \leq i < l_{x+1}.\]

\[\text{2.2 Backward Search}\]

We can answer count queries for a pattern \(P\) of length \(m\) by using a backward search [10]. We search for \(P\) by iterating from right to left over \(P\) and determining in iteration \(i\) (implicitly) the suffix array interval \([b, c]\) that is prefixed by \(P[m-i+1, m]\). We initialize \([b, c]\) with \([1, n]\).

Now, consider the \(i\)-th iteration. Let \(c = P[m-i+1]\) be the character under consideration. We can find the suffix array interval that is prefixed by \(c\) by accessing \(C[c]\). By using the LF-property, we have to jump over all occurrences of \(c\) in \(L\) before \([b, c]\) by using rank queries. So, we set \(b = C[c] + \text{rank}(L, c, b-1) + 1\) and \(e = C[c] + \text{rank}(L, c, e)\). Having read \(P\) completely, the output \([b, c]\) is the number of occurrences of \(P\) in \(T\). If \([b, e]\) = 0, there are no occurrences of \(P\) in \(T\). If the suffix array is at hand, we can also answer locate queries by outputting all starting positions of the occurrences in \(T\).

A main insight of the r-index is the following: If we know the last value of a suffix array interval, we can reconstruct the values in the entire interval without explicitly storing SA. This is formalized by the function \(\Phi\), where \(\Phi(SA[i]) = SA[i-1]\) for \(i > 1\) and \(\Phi(SA[1]) = SA[n]\). The following property holds, which allows us to implement \(\Phi\) in \(O(r)\) space – note the similarity to Lemma 1.

\[\text{Lemma 2 ([11], lemma 3.5). Let } \{u_1, u_2, ..., u_{r+1}\} = \{SA[l_1], SA[l_2], ..., SA[l_r], n + 1\} \text{ and } u_1 < u_2 < ... < u_{r+1} = n + 1. \text{ Then } \Phi(i) = \Phi(u_x) + (i - u_x) \text{ for } u_x \leq i < u_{x+1}.\]

\[\text{3 The Move Data Structure}\]

We now describe the move data structure [19]. By Lemma 1, the LF function can be divided into \(r\) blocks with starting positions \(l_1, l_2, ..., l_r\) and \(l_{r+1} = n + 1\). Similarly, we can divide \(\Phi\) into \(r\) blocks (see Lemma 2) with starting positions \(u_1, u_2, ..., u_r\). Thus, we only need to store the function values for each block head, and can compute every other value by adding the offset between the value and the block head. In the following we will generalize this concept to interval sequences [19] and describe how to efficiently answer queries on interval sequences.
Move-r: Optimizing the r-index

3.1 Disjoint Interval Sequence

We first formally define disjoint interval sequences.

Definition 3. Let \( I = (p_1, q_1), (p_2, q_2), \ldots, (p_k, q_k) \) be a sequence of pairs of elements in the range \([1, n]\), where \(q_1, q_2, \ldots, q_k\) are pairwise distinct, let \( \pi \) be the permutation of \([1, k]\) s.t. \( 1 = q_{\pi[1]} < q_{\pi[2]} < \ldots < q_{\pi[k]} \leq n \), let \( p_{k+1} = n + 1 = q_{k+1} \), and let \( d_i = p_{i+1} - p_i \) for \( i \in [1, k] \). Then, \( I \) is called a disjoint interval sequence if \( 1 = p_1 < p_2 < \ldots < p_k < p_{k+1} \) and \( d_{\pi[i]} = q_{\pi[i]+1} - q_{\pi[i]} \) for all \( i \in [1, k] \).

A disjoint interval sequence defines a sequence of input intervals and output intervals. The input intervals consist of the ranges \([p_i, p_i + d_i]\), and the output intervals consist of the ranges \([q_i, q_i + d_i]\), where the input interval starting with \( p_i \) and the output interval starting with \( q_i \) have the same size \( d_i \). It describes the bijective function \( f_I \) that maps a position in an input interval to the position with the same offset in its corresponding output interval by \( f_I(i) = q_i + (i - p_i) \), where \( x \) is the index of the input interval containing the position \( i \). See Figure 1 for an example. The intervals at the top represent the input intervals and the intervals at the bottom represent the output intervals. A line between an input interval and an output interval is drawn between corresponding intervals and represents the permutation \( \pi \). A move query \( \text{move}(i, x) = (i', x') \) evaluates for a given value \( i \in [1, n] \) and the index \( x \in [1, k] \) of the input interval containing \( i \) the function \( f_I(i) = i' \), and also returns the index \( x' \) of the input interval containing \( i' \). \( x' \) can then be used to directly evaluate the next query \( \text{move}(i', x') \). See again Figure 1 for an example.

To answer move queries on \( I \), we use a move data structure \( M \), which stores three arrays \( M_p[1..k] = [p_1, p_2, \ldots, p_k] \), \( M_q[1..k] = [q_1, q_2, \ldots, q_k] \) and \( M_{\text{idx}}[1..k] \), where \( M_{\text{idx}} \) stores at position \( j \) the index of the input interval containing \( q_j \), i.e., \( M_{\text{idx}}[j] = i \) iff \( q_j \in [p_i, p_i + d_i] \). This allows us to compute the \( i' \)-part of the query \( \text{move}(M, i, x) = (i', x') \) by \( i' = M_q[x] + (i - M_p[x]) \). To find \( x' \), we set \( x' \leftarrow M_{\text{idx}}[x] \). Now, we have to iterate over the starting positions of the input intervals while incrementing \( x' \) to find the input interval containing \( i' \). In the worst case this can take \( O(k) \) steps (see Figure 2). In the following, we show how to reduce the time to answer a move query to \( O(a) \) for a fixed parameter \( a \).

3.2 Balancing a disjoint interval sequence

Now, we show how to balance a disjoint interval sequence, which guarantees constant time to answer move queries.
Definition 4. We call an output interval \([q_j, q_j + d_j]\) of \(I\) \(a\)-heavy if there are at least \(2a\) input intervals of \(I\) starting in it. Otherwise, we call it \(a\)-balanced. We call a disjoint interval sequence \(a\)-balanced if all of its output intervals are \(a\)-balanced. Otherwise, we call it \(a\)-heavy.

The general idea of a balancing algorithm is to find an \(a\)-heavy output interval and split it (and its corresponding input interval) into two new intervals, each. The first new output interval will be \(a\)-balanced, and the second new output interval is either \(a\)-heavy or \(a\)-balanced. By iteratively applying this procedure (the \(a\)-balancing step), we obtain a disjoint interval sequence that is \(a\)-balanced.

Definition 5. We define an \(a\)-balancing step on \(I\) to be the process of splitting an \(a\)-heavy output interval \([q_j, q_j + d_j]\) of \(I\) and its corresponding input interval \([p_j, p_j + d_j]\) at the offset \(d = p_{t+a} - q_j\) to obtain the disjoint interval sequence \(I' = (p_1, q_1), \ldots, (p_j, q_j), (p_j + d, q_j + d_j), (p_{j+1}, q_{j+1}), \ldots, (p_k, q_k)\), where \((p_i, p_i + d_i)\) is the first input interval of \(I\) starting in \([q_j, q_j + d_j]\). We call an algorithm \(\mathcal{A}\) a balancing algorithm, if \(\mathcal{A}\) takes a disjoint interval sequence \(I\) and an integer \(a \geq 2\) as input, iteratively performs \(a\)-balancing steps starting with \(I\) until the disjoint interval sequence is \(a\)-balanced, and finally returns the resulting disjoint interval sequence.

Now we show that every balancing algorithm terminates and that the resulting disjoint interval sequence is at most \(\frac{a}{a-1}\) times as large as \(I\).

Theorem 6 (Generalization of Lemma 6 [19]). Let \(I\) be a disjoint interval sequence of size \(k\), let \(a \geq 2\) be an integer, let \(\mathcal{A}\) be a balancing algorithm and let \(\mathcal{B}_a(I)\) be the disjoint interval sequence resulting from an execution of \(\mathcal{A}\) on \(I\). Then \(f_{\mathcal{B}_a(I)} = f_I\) and \(|\mathcal{B}_a(I)| \leq \frac{a}{a-1}k\).

Proof. Let \(t'\) be the number of \(a\)-balancing steps \(\mathcal{A}\) performs during this execution, let \(t \in [1, t']\), let \(I_t\) be the disjoint interval sequence after the \(t\)-th step, let \(I_0 = I\), let \([q_j, q_j + d_j]\) be the output interval of \(I_{t-1}\) that \(\mathcal{A}\) splits in the \(t\)-th step, and let \([p_i, p_i + d_i]\) be the first input interval of \(I_{t-1}\) starting in \([q_j, q_j + d_j]\).

Since \(\mathcal{A}\) splits \([p_j, p_j + d_j]\) and \([q_j, q_j + d_j]\) at the same offset \(d = p_{t+a} - q_j\) to obtain \(I_t\) from \(I_{t-1}\), clearly \(|I_t| = |I_{t-1}| + 1\) and \(f_I = f_{I_{t-1}}\) and therefore \(|I_t| = k + t\) and \(f_I = f_{I_{t-1}}… = f_{I_t} = f_{\mathcal{B}_a(I)}\).

Let \(O_{a,t}\) be the set of output intervals in \(I_t\) containing at least \(a\) input intervals of \(I_t\). We have \(O_{a,t} \supseteq O_{a,t-1} \setminus \{[q_j, q_j + d_j]\} \cup \{[q_j, q_j + d], [q_j + d, q_j + d_j]\}\), which implies \(|O_{a,t}| \geq |O_{a,t-1}| + 1\) and therefore \(|O_{a,t}| \geq t\). Since the output intervals in \(O_{a,t}\) are pairwise disjoint, there cannot be less than \(a|O_{a,t}|\) input intervals in \(I_t\), which yields \(k + t = |I_t| \geq a|O_{a,t}| \implies t \leq \frac{k}{a-1}\) and therefore \(|\mathcal{B}_a(I)| = k + t' \leq \frac{a}{a-1}k\).

Recall that the dominating part of answering a query \(\text{move}(i, x) = (i', x')\) is to determine \(x'\) by starting at the first input interval starting in the output interval \([q_x, q_x + d_x]\) and iterating over possibly all other input intervals starting in \([q_x, q_x + d_x]\). Since in \(\mathcal{B}_a(I)\), the number of these input intervals is bounded by \(2a - 1\), the following corollary holds.

Corollary 7 (The Move Data Structure). Let \(I\) be a disjoint interval sequence of length \(k\) and \(a \geq 2\) be an integer. Then there is a data structure of size \(O(\frac{a}{a-1})k\) that allows us to consecutively evaluate \(f_I\) in \(O(a)\) time using move queries.
Algorithmic Optimizations

At first, we present our new efficient algorithm to balance disjoint interval sequences. Then, we discuss Move-r and compare it with Nishimoto and Tabei’s OptBWTR [19] index. In Appendix B, we discuss further practical improvements.

4 New algorithm to balance disjoint interval sequences

For constructing an $r$-balanced disjoint interval sequence given a disjoint interval sequence, we use two balanced search-trees $T_{in}$ and $T_{out}$ storing the pairs of $I$ sorted by the starting positions of the input intervals and the output intervals, respectively. We add $(n+1, n+1)$ to both $s.t.$ for every pair in $T_{in}$ or $T_{out}$ we can compute the length of the input- or output-interval it creates by accessing the next pair in $T_{in}$ or $T_{out}$, respectively.

To speed up the construction of $T_{out}$, we at first calculate the permutation $\pi[1..k]$ from the definition of the disjoint interval sequence by sorting any permutation of $[1, k]$ by the starting positions of the output intervals of $I$. Since $\pi[i]$ points to the pair creating the $i$-th output interval, we can build $T_{out}$ in $\mathcal{O}(k)$ additional time by inserting the pairs in the order $(p_{\pi[1]}, q_{\pi[1]}), (p_{\pi[2]}, q_{\pi[2]}), \ldots, (p_{\pi[k]}, q_{\pi[k]})$ into $T_{out}$.

Let $T$ be a balanced search-tree over a set $X$. Then $\langle v \rangle$ denotes the node in $T$ storing $v \in X$, $T$.has-next($\langle v \rangle$) returns whether $\langle v \rangle$ has a successor node in $T$, $T$.next($\langle v \rangle$) returns this successor node, $T$.next($\langle v \rangle$, $n$) returns the $n$-th successor of $\langle v \rangle$ in $T$, $T$.pred($\langle v \rangle$) = $\langle \max\{w \in T \mid w \leq v\} \rangle$, $T$.succ($\langle v \rangle$) = $\langle \min\{w \in T \mid v \leq w\} \rangle$, $T$.min() = $\langle \min\{v \in T\} \rangle$ and $\langle p, q \rangle_{in}$ and $\langle p, q \rangle_{out}$ denote the nodes in $T_{in}$ and $T_{out}$ storing $(p, q)$, respectively. The algorithm is-a-heavy expects an output interval $[q_j, q_j + d_j]$ and $\langle p_i, q_i \rangle_{in}$, where $[p_i, p_i + d_i]$ is the first input interval starting in $[q_j, q_j + d_j]$. It iterates forward at most $2a$ steps in $T_{in}$ starting with $\langle p_i, q_i \rangle_{in}$ and returns whether $[q_j, q_j + d_j]$ is a-heavy.

4.1.1 Framework Balancing Algorithm

The general idea of our balancing algorithm is to not explicitly store all $a$-heavy output intervals in a data structure, but to compute and balance them on-the-fly. We achieve this by implementing a deterministic balancing algorithm that in each step chooses to balance the output interval with the smallest starting position. Algorithm 1 is the framework of our balancing algorithm.

In the iterations of main while-loop (lines 3-13), we consider the pairs $(p_i, q_i) \in T_{in}$ and $(p_j, q_j) \in T_{out}$ ascendingly ordered by $q_j$, where $i$ is minimal s.t. $q_j \leq p_i < q_j + d_j$ holds, if such $i$ exists, i.e., we consider the output intervals $[q_j, q_j + d_j]$ in ascending order of their starting position together with the respective first input intervals $[p_i, p_i + d_i]$ starting in them. We initialize $\langle p_i, q_i \rangle_{in}$ and $\langle p_j, q_j \rangle_{out}$ with the first pairs in $T_{in}$ and $T_{out}$, respectively. For those, $q_j \leq p_i < q_j + d_j$ trivially holds, since $q_1 = p_1 = 1 < q_1 + d_1$.

Given some combination of such pairs $\langle p_i, q_i \rangle_{in}$ and $\langle p_j, q_j \rangle_{out}$, we want to find the next combination of such pairs, that is we want to find $(p_{i'}, q_{i'}) \in T_{in}$ and $(p_{j'}, q_{j'}) \in T_{out}$, where $q_{i'}$ is minimal s.t. $q_{i'} > q_j$ and there exists a pair $(p_{i'}, q_{i'}) \in T_{in}$, where $p_{i'}$ is minimal s.t. $q_{j'} \leq p_{i'} < q_{j'} + d_{j'}$. To find those, we at first iterate further one step with $\langle p_j, q_j \rangle_{out}$ in $T_{out}$ (line 8). We then iterate (possibly zero steps) with $\langle p_i, q_i \rangle_{in}$ in $T_{in}$, until $p_i > q_j$ holds, or we have reached the second to last pair in $T_{in}$ (lines 10-12). If then also $p_i < q_j + d_j$ holds (line 13), we have found the next combination of such pairs. Else, we again alternately iterate further as described with $\langle p_i, q_i \rangle_{in}$ and $\langle p_j, q_j \rangle_{out}$ until we find such pairs or until we reach the second to last pair in $T_{in}$ or $T_{out}$, respectively.
We want to make sure that each output interval \([q_j, q_j + d_j]\) considered in the main while-loop is \(a\)-balanced. To do this, we check if \([q_j, q_j + d_j]\) is \(a\)-heavy by calling \texttt{is-a-heavy}([q_j, q_j + d_j], \langle \pi, \eta \rangle) in line 4. If it is \(a\)-balanced, we proceed to iterate further over \(\mathcal{T}_\pi\) and \(\mathcal{T}_\mathcal{E}\) as described to find the next combination of \(\langle \pi, \eta \rangle\) and \(\langle \pi, q_j \rangle\) out (lines 7-13). Else, we call \texttt{balance-up-to}(\langle \pi, \eta \rangle, \langle \pi, q_j \rangle) in line 5, which performs at least one \(a\)-balancing step. More precisely, it splits \([q_j, q_j + d_j]\) at the offset \(d = p_i + a - q_j\) into two new output intervals \([q_j, q_j + d]\) and \([q_j + d, q_j + d]\), then possibly recursively \(a\)-balances all output intervals starting before \(q_j\) becoming \(a\)-heavy in the process. We will prove the correctness of \texttt{balance-up-to} in the next section (see Theorem 9). After each iteration of the main while-loop, all output intervals starting before \(q_j\) are \(a\)-balanced, hence, after the main while-loop, all output intervals are \(a\)-balanced.

### 4.1.2 Balancing Step

In Algorithm 1, we always call \texttt{balance-up-to} with \(q_j = q_u\) (the non-recursive case). Inside \texttt{balance-up-to}, we will (possibly) recursively call \texttt{balance-up-to} with \(q_j < q_u\) only (the recursive case). At first, we prove the correctness of \texttt{balance-up-to} in the recursive case. Let \(t'\) denote the number of balancing steps our balancing algorithm performs.

\begin{algorithm}[H]
\caption{balance-framework()}
\begin{algorithmic}[1]
\State \((\pi, \eta)\) \(<-\mathcal{T}_\pi.\text{min}();
\State \((\pi, q_j)\) \(<-\mathcal{T}_\mathcal{E}.\text{min}();
\While {true}
\If {is-a-heavy([q_j, q_j + d_j], \langle \pi, \eta \rangle)}
\State balance-up-to(\langle \pi, \eta \rangle, \langle \pi, q_j \rangle);
\Else
\State \langle \pi, q_j \rangle \(<-\mathcal{T}_\mathcal{E}.\text{next}([\pi, q_j]);
\If {\neg \mathcal{T}_\mathcal{E}.\text{has-\text{next}}([\pi, q_j])} \Return ;
\While {\pi < q_j}
\State \langle \pi, q_j \rangle \(<-\mathcal{T}_\pi.\text{next}([\pi, q_j]);
\If {\neg \mathcal{T}_\pi.\text{has-\text{next}}([\pi, q_j])} \Return ;
\While {q_j + d_j < \pi};
\EndWhile
\EndWhile
\EndIf
\EndWhile
\EndIf
\EndWhile
\EndIf
\EndAlgorithm
\end{algorithm}

\begin{algorithm}[H]
\caption{balance-up-to(\langle \pi, \eta \rangle, \langle \pi, q_j \rangle, q_u).}
\begin{algorithmic}[1]
\State \langle \pi+i, a, q+i+a \rangle \(<-\mathcal{T}_\pi.\text{next}([\pi, \eta]);
\State \(d\) \(<-\pi+i-a-q_j;
\State \mathcal{T}_\pi.\text{insert}([p_j + d, q_j + d]);
\State \mathcal{T}_\mathcal{E}.\text{insert}([p_j + d, q_j + d]);
\State \If {p_j + d < q_u}
\State \langle q_j, q_j \rangle \(<-\mathcal{T}_\mathcal{E}.\text{pred}(p_j + d);
\State \langle q_j, q_j \rangle \(<-\mathcal{T}_\pi.\text{succ}(q_j);
\State \If {is-a-heavy([q_j, q_j + d_j], \langle p_j, q_j \rangle)}
\State balance-up-to(\langle p_j, q_j \rangle, \langle q_j, q_j \rangle, q_u);
\State \EndIf
\State \EndIf
\State \EndAlgorithm
\end{algorithm}

1:8 Move-r: Optimizing the r-index

[Diagram]

**Figure 3** Illustration of an a-balancing step, where \( p_j + d < q_u \), and \( O_u = [q_y, q_u + d_u] \) is a-heavy.

**Lemma 8.** Let \( s \in [0, t') \), let \([q_j, q_j + d_j]\) be an a-heavy output interval of \( I_s \) and let \( q_u > q_j \), where \([q_j, q_j + d_j]\) contains exactly 2a input intervals of \( I_s \), and all other output intervals of \( I_s \) starting before \( q_u \) are a-balanced. Let \([p_i, p_i + d_i]\) be the first input interval of \( I_s \) starting in \([q_j, q_j + d_j]\), let \( t \in (s, t') \) be minimal s.t. all output intervals in \( I_t \) starting before \( q_u \) are a-balanced, and assume \( \mathcal{T}_{in} \) and \( \mathcal{T}_{out} \) store \( I_s \). Then calling balance-up-to(\(\langle p_{i+a}, q_{i+a}\rangle_{in}\), \(\langle p_j, q_j\rangle_{out}, q_u\)) updates \( \mathcal{T}_{in} \) and \( \mathcal{T}_{out} \) to store \( I_t \).

**Proof.** We at first update \( \mathcal{T}_{in} \) and \( \mathcal{T}_{out} \) to store \( I_{s+1} \) by finding \(\langle p_{i+a}, q_{i+a}\rangle_{in}\) and inserting \(p_j + d, q_j + d\) into \(\mathcal{T}_{in}\) and \(\mathcal{T}_{out}\), where \(d = p_{i+a} - q_j\). Compared with \(I_{s+1}\), there are two new output intervals \([q_j, q_j + d]\) and \([q_j + d, q_j + d_j]\) in \(I_{s+1}\). Both contain exactly a input intervals, hence they are a-balanced. Let \(O_y = [q_y, q_y + d_y]\) be the output interval of \(I_{s+1}\) containing \(p_j + d\). Since \(p_j + d \in O_y\), \(O_y\) is the only possibly a-heavy output interval in \(I_{s+1}\) starting before \(q_u\). If the if-clause in line 5 fails, then \(O_y\) starts at or after \(q_u\), hence \(I_t = I_{s+1}\) and we are done.

Else, let \(p_z, p_z + d_z\) be the first input interval of \(I_{s+1}\) starting in \(O_y\). We now find \(\langle p_y, q_y\rangle_{out}\) with a predecessor search over \(\mathcal{T}_{out}\) and \(\langle p_j, q_j\rangle_{in}\) using a successor search over \(\mathcal{T}_{in}\). Then, we check if \(O_y\) is a-heavy in \(I_{s+1}\) by calling is-a-heavy(\(\langle p_y, q_y + d_y\rangle, \langle p_j, q_j\rangle_{in}\)). If it is a-balanced, then \(I_t = I_{s+1}\) and we are done.

If it is a-heavy, then there are exactly 2a input intervals of \(I_{s+1}\) starting in \(O_y\), because \(O_y\) is a-balanced in \(I_s\) and compared with \(I_{s+1}\), the number of input intervals of \(I_{s+1}\) starting in \(O_y\) has increased by exactly one (see Figure 3). Therefore, calling balance-up-to(\(\langle p_z, q_z\rangle_{in}\), \(\langle p_y, q_y\rangle_{out}, q_u\)) in line 9 satisfies the requirements of Lemma 8, hence we can iteratively apply the same overall argument to obtain \(I_{s+2}\) from \(I_{s+1}\), \(I_{s+3}\) from \(I_{s+2}\), etc. in the inner recursive calls of balance-up-to. Let balance-up-to(\(\langle p_{j'}, q_{j'}\rangle_{in}\), \(\langle p_{j'}, q_{j'}\rangle_{out}, q_u\)) be the \(t - s - 1\)-th such recursive call and let \(O_{y'} = [q_{y'}, q_{y'} + d_{y'}]\) be the output interval of \(I_t\) containing \(p_{j'} + d'\), where \(d' = p_{j'} + a - q_j\). In this call, we update \(\mathcal{T}_{in}\) and \(\mathcal{T}_{out}\) to store \(I_t\). By the definition of \(I_t\), \(O_{y'}\) either starts at or after \(q_u\) or is a-balanced in \(I_t\), hence one of the if-clauses fails and the series of recursive calls ends. \(\blacksquare\)

With Lemma 8, we can now prove the correctness of balance-up-to in the non-recursive case \((q_j = q_u)\).

**Theorem 9.** Let \(s \in [0, t')\), let \([q_j, q_j + d_j]\) be the first a-heavy output interval of \(I_s\) and let \([p_i, p_i + d_i]\) be the first input interval of \(I_s\) starting in \([q_j, q_j + d_j]\). Let \(t \in (s, t')\) be minimal s.t. all output intervals in \(I_t\) starting before \(q_j\) are a-balanced, and assume \(\mathcal{T}_{in}\) and \(\mathcal{T}_{out}\) store \(I_s\). Then calling balance-up-to(\(\langle p_i, q_i\rangle_{in}\), \(\langle p_j, q_j\rangle_{out}\)) updates \(\mathcal{T}_{in}\) and \(\mathcal{T}_{out}\) to store \(I_t\).

**Proof.** As in the proof of Lemma 8, let \(O_y = [q_y, q_y + d_y]\) be the output interval of \(I_{s+1}\) containing \(p_j + d\). If \(O_y\) starts at or after \(q_j\) or if \(O_y\) is a-balanced in \(I_{s+1}\), then \(I_t = I_{s+1}\) and we are done after inserting \((p_j + d, q_j + d)\) into \(\mathcal{T}_{in}\) and \(\mathcal{T}_{out}\) (lines 3 and 4).
Else we can argue as in the proof of Lemma 8 to infer that $O_y$ is the only $a$-heavy output interval of $I_{s+1}$ starting before $q_j$, there are exactly $2a$ input intervals of $I_{s+1}$ starting in $O_y$ and that we call balance-up-to($p_2, q_z$)$_{in}$, ($p_y, q_y$)$_{out}$, $q_j$) in line 12, which updates $T_m$ and $T_{out}$ to store $I_t$ by Lemma 8.

4.1.3 Running time analysis

Theorem 10. Given $T_m$ and $T_{out}$ for a disjoint interval sequence $I$ of size $k$ and an integer $a \geq 2$, we can compute $B_a(I)$ in $O(k + \frac{k}{a} \log k)$ time and $O(1)$ additional space.

Proof. As argued in Section 3.2, $T_m$ contains such an $a$-balanced disjoint interval sequence $B_a(I)$ after performing Algorithm 1. Here, we iterate once over $T_m$ and $T_{out}$ in total. This takes $O(k)$ time. Overall, we call balance-up-to (recursively and non-recursively) at most $\frac{k}{a-1}$ times, since by Theorem 6, the disjoint interval sequence is $a$-balanced after $\frac{k}{a-1}$ balancing steps. In Algorithm 1, we need $O(1)$ additional space.

In each of these calls of balance-up-to, we find $\langle p_{i+a}, q_{j+a} \rangle$ in $O(a)$ time, insert $(p_j + d, q_j + d)$ into $T_m$ and $T_{out}$ ($O(\log k)$ time) and possibly perform a successor and a predecessor search over $T_{out}$ and $T_m$ ($O(\log k)$ time), respectively. Overall, one call of balance-up-to needs $O(a + \log k)$ time. We do not need the local variables in balance-up-to, hence balance-up-to needs $O(1)$ additional space. In total, this yields $O(k + \frac{k}{a}(a + \log k)) = O(k + \frac{k}{a} \log k)$ time and $O(1)$ additional space.

4.2 Practically optimized locate algorithm

Now we describe our practically optimized locate algorithm. At first, we introduce preparatory definitions and lemmas. Then, we discuss Nishimoto and Tabei’s count algorithm [19] together with our practically optimized version of their locate algorithm.

Definition 11. Let $M^L_\Phi$ of size $r'$ $\geq r$ be a move data structure (Corollary 7) built from the disjoint interval sequence $B_a(I_{LF})$, where $I_{LF} = (l_1, LF(l_1)), (l_2, LF(l_2)), ..., (l_r, LF(l_r))$. Let $M^L_\Phi$ be a move data structure built from the disjoint interval sequence $B_a(I_{LF})$, where $I_{LF} = (l_1, \Phi(u_1)), (l_2, \Phi(u_2)), ..., (l_r, \Phi(u_r))$. Let $L'[1..r']$ be a string containing the characters $L[M^L_\Phi[1]], L[M^L_\Phi[2]], ..., L[M^L_\Phi[r']]$.

Note that $r'$ and $r''$ depend on the balancing parameter $a$. The total used space of these data structures is $O(r)$, because $r', r'' \leq \frac{a}{a-1} r = O(r)$ hold with Theorem 6.

Definition 12. Let $i \in [1, m]$, let $[b_i, e_i]$ be the suffix array interval of $P_i$, let $b'_i = LF^{-1}(b_i)$, $e'_i = LF^{-1}(e_i)$, let $\hat{b}_i, \hat{e}_i, \hat{b}'_i$ and $\hat{e}'_i$ be the indices of the input intervals in $M^L_\Phi$ containing $b_i, e_i, b'_i$ and $e'_i$, respectively. Define $z(i) = SA[e_i]$ (the lexicographically largest suffix of $T$ starting with $P_i$), let $s(i, j) = z(j) + i - j$ and let $y(i) = \max\{ j \in [1, m] | T_{y(i)}(j) \text{ starts with } P_i \}$. Finally, define $b_m+1, b_m+1, e_m+1$ and $\hat{e}_m+1$ analogously, i.e., $b_m+1 = 1, b_m+1 = 1, e_m+1 = n$ and $\hat{e}_m+1 = r'$.

Let $S$ be a string that occurs in $T$ and let $T_i$ be the lexicographically largest suffix of $T$ among those starting with $S$. Then we call $T[1..l + |S|]$ the lexicographically largest occurrence of $S$ in $T$. Intuitively, $P_{y(i)}$ is the shortest suffix of $P$ s.t. the lexicographically largest occurrence of $P_{y(i)}$ in $T$ is a suffix (in $T$) of the lexicographically largest occurrence of $P_i$ in $T$.

Lemma 13. For $i \in [1, m]$, $T_{s(i, y(i))}$ is the lexicographically largest suffix of $T$ starting with $P_i$, that is, it holds $s(i, y(i)) = z(i)$. 

Sea 2024
\textbf{Move-r: Optimizing the r-index}

**Proof.** Suppose there was a suffix $T_w > T_{z(y(i))}$ starting with $P_i$. Then $T_{w+y(i)-1}$ starts with $P_{y(i)}$ and $T_{w+y(i)-1} = T_{z(y(i))}$, contradicting the definition of $T_{z(y(i))}$.

\textbf{Lemma 14.} For $i \in [1, m]$, $p = \text{SA}^{-1}[z(y(i)) + 1]$ is the end position of a BWT run.

**Proof.** See Appendix A.

The query $\text{select}(T, c, i)$ returns the position of the $i$-th occurrence of $c$ in $T$.

\textbf{Lemma 15.} Let $i \in [1, m]$. Given $b_{i+1}, \hat{b}_{i+1}, e_{i+1}$ and $\hat{e}_{i+1}$ (and additionally $y(i+1)$ and $e'_{y(i+1)}$, for $i < m$). If we augment $L'$ with an $O(\log \log \sigma)$ time and $O(r')$ space rank data structure and an $O(1)$ time and $O(r')$ space select data structure, we can compute $b_i, \hat{b}_i, e_i, \hat{e}_{i}$, $y(i)$ and $e'_{y(i)}$ in $O(\alpha + \log \log \sigma)$ time.

**Proof.** Let $c = P[i]$. To obtain $b_i$ and $\hat{b}_i$ from $b_{i+1}$ and $\hat{b}_{i+1}$, we can use that $T_{\text{SA}[b_i]}$ is the lexicographically smallest suffix of $T$ starting with $P_{i+1}$ that is preceded by $c$, and therefore, $b_i'$ and $\hat{b}_i'$ are the first occurrences of $c$ at or after $b_{i+1}$ and $\hat{b}_{i+1}$ in $L$ and $L'$, respectively, allowing us to compute $b_i' = \text{select}(L', c, \text{rank}(L', b_{i+1} - 1) + 1)$, $b_i' =  \mathcal{M}_p[L_i']$ and $(b_i, \hat{b}_i) = \text{move}(\mathcal{M}_p[L_i], b_i', \hat{b}_i')$. Analogously, we obtain $e_i$ and $\hat{e}_i$ from $e_{i+1}$ and $\hat{e}_{i+1}$ by $e_i' = \text{select}(L', c, \text{rank}(L', \hat{e}_{i+1}))$, $e_i' =  \mathcal{M}_p[L_i']$ and $(e_i, \hat{e}_i)$ = $\text{move}(\mathcal{M}_p[L_i], e_i', \hat{e}_i')$. An implementation of the required rank-select data structure for $L'$ can be found in [19] or Appendix B.1.

Now it remains to compute $y(i)$ and $e'_{y(i)}$. If $i = m$, then $y(i) = m$, because $P_m$ occurs at $z(m)$, and therefore $e'_{y(i)} = e'_m$. Else, we have $i < m$. Here, we consider two cases (see Figure 4).

**Case 1:** $e_i' = e_{i+1}$. Then $T_{\text{SA}[e_{i+1}]}$ is preceded by $c$, hence $P_i$ occurs at $y(i) - 1 \overset{\text{Loc.}}{=} s(i+1, y(i)) - 1 \overset{\text{Def.}}{=} s(i, y(i))$, hence (i) $y(i) \geq y(i+1)$. Let $j \in (y(i+1), m]$. Since $P_{i+1}$ does not occur at $s(i+1, j)$, $P_i$ does not occur at $s(i+1, j) - 1 \overset{\text{Def.}}{=} s(i, j)$, hence (ii) $y(j) \leq y(i+1)$. Combining (i) and (ii) yields $y(i) = y(i+1)$, and therefore $e'_{y(i)} = e'_{y(i+1)}$.
Case 2: \( e_i' < e_{i+1} \). Then \( T_{i+1(j)} \) is not preceded by \( e_i \), hence \( P_i \) does not occur at \( z(i + 2) - 1 \).

By induction, \( P_i \) also does not occur at \( z(i + 2) - 2, z(i + 3) - 3, \ldots, z(m + i - m \). More generally, \( P_i \) does not occur at \( s(i, j) \), for each \( j \in (i, m) \), hence (ii) \( y(i) \geq i \). Since \( P_i \) occurs at \( z(i) \), we get (ii) \( y(i) \geq i \). Combining (i) and (ii) yields \( y(i) = i \), and therefore \( \hat{e}_i' = \hat{e}_i \).

In total, we perform two move queries on \( \mathcal{M}^\Phi(\mathcal{O}(a) \text{ time}) \) and at most two rank queries on \( L' \) \( (\mathcal{O}(\log \log \sigma) \text{ time}) \). The remaining computations take \( \mathcal{O}(1) \) time, hence the running time bound follows.

Note that iff \( L'[\hat{b}_i + 1] = c \) holds, then \( b_i' = b_i + 1 \) and \( \hat{b}_i' = \hat{b}_i + 1 \). Similarly, \( L'[\hat{e}_i + 1] = c \Leftrightarrow e_i' = e_i + 1 \land \hat{e}_i' = \hat{e}_i + 1 \). In practice, we use this to save up to two rank-select queries on \( L' \), which improves performance, especially if the text has a small alphabet. This optimization and the first part of Lemma 15, i.e., the computation of \( b_i, e_i, \hat{b}_i \) and \( \hat{e}_i \) is identical in the algorithms proposed by Nishimoto and Tabei [19].

▶ Theorem 16. We can answer a count query in \( \mathcal{O}(m \log \log \sigma) \) time.

Proof. We compute \( b_1, e_1, \hat{b}_1 \) and \( \hat{e}_1 \) by applying Lemma 15 \( m \) times starting with \( b_{m+1}, e_{m+1}, \hat{b}_{m+1} \) and \( \hat{e}_{m+1} \) and return the length \([b_1, e_1]\) of the suffix array interval of \( P \).

Note that we do not need \( y(1) \) and \( \hat{e}'_{y(1)} \) to answer a count query. Those values are only necessary for answering a locate query, which we will discuss next.

▶ Definition 17. For \( i \in [1, r'] \), let \( p_i = \mathcal{M}^\Phi[p_i] = \mathcal{M}^\Phi[i + 1] - 1 \). If \( p_i \) is the end position of a BWT run, then there exists an output interval in \( L' \) (and therefore also in \( \mathcal{M}^\Phi \)) that starts with \( \text{SA}[p_i] \). Let \( x_i \) be the index of this output interval of \( \mathcal{M}^\Phi \), i.e., \( x_i \in [1, r'] \) s.t. \( \text{SA}[x_i] = \text{SA}[p_i] \). Finally, let \( \text{SA}_\Phi[1..r'] \) be an array, where

\[
\text{SA}_\Phi[i] = \begin{cases} x_i & \text{if } p_i \text{ is the end position of a BWT run,} \\ \bot & \text{else.} \end{cases}
\]

▶ Theorem 18. We can answer a locate query in \( \mathcal{O}(m \log \log \sigma) + \mathcal{O}(\log \log \sigma - a) \) time.

Proof. As in Theorem 16, we compute \( b_1, e_1, \hat{b}_1 \) and \( \hat{e}_1 \) and additionally \( y(1) \) and \( \hat{y}'_{y(1)} \) by applying Lemma 15 \( m \) times starting with \( b_{m+1}, e_{m+1}, \hat{b}_{m+1} \) and \( \hat{e}_{m+1} \). This takes \( \mathcal{O}(m(\log \log \sigma)) \) time. Then, we compute

\[
s = \text{SA}[e_1] \overset{\text{Lem.}}{=} 13 s(1, y(1)) \overset{\text{Def.}}{=} 12 z(y(1)) - y(1) \overset{\text{Def.}}{=} 12 \text{SA}[\hat{e}'_{y(1)}] - y(1)
\]

and report \( s \). Lemma 14 yields, that \( \hat{e}'_{y(1)} \overset{\text{Def.}}{=} \text{LF}^{-1}(\hat{e}_{y(1)}) \overset{\text{Def.}}{=} \text{SA}^{-1}[z(y(1)) + 1] \) is the end position of a BWT run. Since the BWT run end positions are a subset of the input interval end positions of \( \mathcal{M}^\Phi \), \( \hat{e}'_{y(1)} \) is also the end position of the \( \hat{e}'_{y(1)} \)-th input interval of \( \mathcal{M}^\Phi \), hence (i) holds and (ii) follows with Definition 17. If \( b_1 = e_1 \), then we are done.

Else, we compute the remaining occurrences \( \{\text{SA}[e_1 - 1], \text{SA}[e_1 - 2], \ldots, \text{SA}[b_1]\} = \{\Phi^1(s), \Phi^2(s), \ldots, \Phi^{e_1-b_1}(s)\} \) by consecutively performing \( e_1 - b_1 \) move queries on \( \mathcal{M}^\Phi \) starting with \( (s, \hat{s}) \) and reporting \( s \) after each move query, where \( \hat{s} \) is the index of the input interval in \( \mathcal{M}^\Phi \) containing \( s \). This takes \( \mathcal{O}(\log \log \sigma) \) additional time. To compute \( \hat{s} \), we at first compute the index \( \hat{x} = \text{SA}_\Phi[\text{SA}_\Phi[\hat{e}'_{y(1)}]] \) of the input interval in \( \mathcal{M}^\Phi \) containing \( \text{SA}[\hat{e}'_{y(1)}] \). Again, we can
1:12 Move-r: Optimizing the r-index

do this by Definition 17, because \( e'_{y(1)} \) is the end position of both a BWT run and the \( e'_{y(1)} \)th input interval of \( \mathcal{M}^{LF} \). Then, we find \( s \) in \( O(\log m) \) time with an exponential search to the left over the input interval starting positions of \( \mathcal{M}^{\Phi} \) starting at \( \hat{x} \). We can do this, because

\[
\hat{x} - \hat{s} \leq |SA[e'_{y(1)}]| - |SA[e_1]| \quad \text{(Def. 12, Lem. 13)}
\]

\[
z(y(1)) + 1 - s(1, y(1)) \leq m,
\]

where (iii) holds, because an input interval of \( \mathcal{M}^{\Phi} \) has at least size one.

\[\Box\]

Comparing Move-r with OptBWTR

Move-r consists of the data structures \( \mathcal{M}^{LF}, \mathcal{M}^{\Phi}, L' \) (see Definition 11), a rank-select data structure \( RS_L' \) for \( L' \) (see Appendix B.1) and \( SA_{\Phi} \) (see Definition 17). Overall, Move-r needs \( O(r) \) space, because \( r', r'' = O(r) \) hold with Theorem 6.

Now we discuss the differences between OptBWTR and Move-r. OptBWTR implements \( \Phi^{-1} \) instead of \( \Phi \), which results in symmetric algorithms and data structures. To compute the initial suffix array value during a locate query, OptBWTR stores two arrays \( SA_a[1..r'] \) and \( SA_{dek}[1..r'] \) instead of \( SA_{\Phi} \), where \( SA_a[i] = SA[\mathcal{M}^{LF}_p][i] \) and \( SA_{dek}[i] \) stores the index of the input interval in \( \mathcal{M}^{\Phi} \) containing \( SA_a[i] \). With Move-r, we access \( SA_{\Phi} \) only once and \( \mathcal{M}^{\Phi} \) at most \( \log m \) times to compute the initial suffix array value during a locate query. This saves up to \( 3m \) cache misses compared with OptBWTR, because there, \( \mathcal{M}^{\Phi}_p \), \( SA_a \) and \( SA_{dek} \) are randomly accessed in potentially each iteration of the backward search. Another difference is that Move-r uses a practically optimized implementation of \( RS_L' \) (see Appendix B.1). Finally, Move-r incorporates further practical optimizations (see Appendix B.2 and Appendix B.3).

5 Experimental Evaluation

We implemented the algorithms described in Section 4 in C++20. We implemented a practically optimized version the balancing algorithm described in Section 4.1. For \( T_m \) and \( T_{out} \), we used the B-tree implementation from the abseil-cpp library. For sorting, we used the in-place sample sort implementation ips40 [1]. We used the uncompressed bit vector- and sd-array implementations in the SDSL [12] to implement \( RS_L' \). To measure peak memory consumption, we used malloc_count. Links to all software used can be found in our GitHub repository.

Now we discuss the tested indexes. Some of them use Big-BWT [15], which constructs the so-called prefix free parsing (PFP) [4] of \( T \) to build the BWT and suffix array samples. This approach reduces the working space needed to construct an r-index from \( O(n) \) to \( O(|PFP|) \) words, where \( |PFP| \) is the sum of the lengths of all dictionary phrases and the number of all phrases in the factorization of the PFP of \( T \). grlBWT [9] is a BWT construction algorithm for string collections that uses string compression. For most inputs, it is currently the fastest and most memory-efficient algorithm. However, it does not support the computation of suffix-array samples, which are necessary for efficiently answering locate queries.

- \texttt{move-r}: Our implementation. It uses Big-BWT build the BWT and suffix array samples.
  - In preliminary experiments, we observed that the balancing process can drastically increase query throughput and determined \( a = 8 \) to be the optimal trade-off between index size and performance.
- \texttt{r-index}: [11] The original implementation of the r-index (adjusted to use Big-BWT).
- \texttt{online-rlbwt}: [2] A refined version of the dynamic RLBWT implementation from [22], which additionally supports answering locate queries.
- \texttt{rcomp-glfig}: [18] A dynamic r-index developed by the inventors of the move data structure. It uses their so-called divided BWT (DBWT) and grouped LF-interval graph (GLFIG) representations of the BWT, which maintain the stricter so-called \( \alpha \)-balancedness property, where \( \alpha \geq 2 \) is an integer. The grouping parameter \( g \) has been set to 16.
Table 1: Statistics of the tested texts ($r'$ and $r''$ calculated for $a = 2$). $N$ denotes the number of queried patterns, $m$ is the pattern length and $occ$ is the average number of occurrences per pattern.

<table>
<thead>
<tr>
<th>text</th>
<th>size [GB]</th>
<th>$\sigma$</th>
<th>n/r</th>
<th>$r'/r$</th>
<th>$r''/r$</th>
<th>$N$</th>
<th>$m$</th>
<th>$occ$</th>
</tr>
</thead>
<tbody>
<tr>
<td>einstein.en.txt</td>
<td>0.47</td>
<td>139</td>
<td>1611.18</td>
<td>1.23</td>
<td>1.49</td>
<td>100000</td>
<td>800</td>
<td>748.82</td>
</tr>
<tr>
<td>sars2</td>
<td>84.19</td>
<td>80</td>
<td>686.57</td>
<td>1.38</td>
<td>1.06</td>
<td>1000</td>
<td>50</td>
<td>52438.82</td>
</tr>
<tr>
<td>dewiki</td>
<td>68.72</td>
<td>210</td>
<td>345.80</td>
<td>1.23</td>
<td>1.35</td>
<td>2000</td>
<td>320</td>
<td>313.49</td>
</tr>
<tr>
<td>chr19</td>
<td>58.57</td>
<td>52</td>
<td>272.20</td>
<td>1.06</td>
<td>2.00</td>
<td>150</td>
<td>10</td>
<td>216783.43</td>
</tr>
<tr>
<td>english</td>
<td>2.21</td>
<td>239</td>
<td>3.36</td>
<td>1.19</td>
<td>1.20</td>
<td>30000</td>
<td>35</td>
<td>34.60</td>
</tr>
</tbody>
</table>

For reasons of completeness, we also include the following data structures in our test, although they do not support locate queries. They should thus not be directly compared with the other indexes.

- **r-index-f**: [5] An r-index that also uses a move data structure to compute LF. However, it does not perform a balancing algorithm on $I_{LF}$. It uses Big-BWT and pfp-thresholds, to build the BWT, and has been optimized to reduce the index size. It does not support answering locate queries. We used the variant lookup-bv as recommended by the authors.

- **block-rlbwt-2**: [8] An index that splits $L$ into blocks of size $b = 2^{11}$ and applies run-length encoding to those. $\text{rank}(L, c, i)$ is computed by looking up the number of occurrences of $c$ before the $\lceil i/b \rceil$-th block and scanning over it up to position $i$. Runs exceeding length $2^{10-\left\lceil \log_2 \sigma \right\rceil}$ are split s.t. one run can be encoded using two bytes. It uses grlBWT to build the BWT and does not support answering locate queries.

- **block-rlbwt-v**: [8] Uses the same approach as block-rlbwt-2, but avoids splitting runs by using $O(l + \log \sigma)$ bytes to encode a run of length $l$, and uses $b = 2^{14}$.

- **block-rlbwt-r**: [8] Uses the same approach as block-rlbwt-2, but instead splits $L$ into blocks of $b' = 32$ runs. To compute $\text{rank}(L, c, i)$, the block containing $i$ is found using a heap ordered B$^+$-tree of the block starting positions.

We compared all indexes using our tool move-r-bench, which is included in our GitHub repository, on a system with two AMD EPYC 7452 CPUs (32/64x 2.35-3.35GHz, 2/16/128MB L1/2/3 cache) and 1TB of 3200 MT/s DDR4 RAM using the GCC 9.4.0 compiler and the compile flags "-march=native -DNDEBUG -Ofast" on Ubuntu 18.04.6. Table 1 shows the tested texts. einstein.en.txt and english are part of the Pizza&Chili Corpus. dewiki is a highly repetitive text that has been handcrafted from German Wikipedia entries. chr19 consists of concatenated human chromosome 19 haplotypes, and sars2 is a collection of Sars-Cov-2 genomes, both of which were crafted out of datasets from the National Center for Biotechnology Information (NCBI) database. Links to all texts can be found in our GitHub repository.

5.1 Construction performance

Figure 5 shows index construction performance (time and peak memory consumption). Commonly, the block-rlbwt indexes can be constructed the fastest and while using the least space (except with einstein.en.txt). This is because they use grlBWT to build the BWT.
Move-r: Optimizing the r-index

With repetitive texts, index construction time and peak memory usage are dominated by the construction of the BWT. In this case, any static r-index can easily be adapted to instead use gr1BWT and achieve similar construction performance with the limitation that it then only supports count queries. Compared with the closest other index supporting locate queries, move-r can be constructed 0.9–2 (typically 2) times as fast while requiring 1/3–1 (typically 1/2) times as much memory. In some cases, move-r even competes with the dynamic indexes regarding memory usage (see chr19, dewiki and english). Constructing static indexes is 2–20 times as fast and requires 1–10 times as much space than constructing dynamic indexes. Comparing the two dynamic indexes, rcomp-glfig's construction consumes 2–3 times as much memory as that of online-rlbwt while sometimes one and sometimes the other takes (at most 60%) longer.

Figure 5 Peak memory consumption during the index construction versus index construction time. The indexes in the left legend column (solid marks) support answering locate queries, while the others do not.
Figure 6 Query throughput versus index size. The indexes in the upper legend row (solid marks) support answering locate queries, while the others do not.
5.2 Query performance

For each text, we generated two sets of query patterns (two lines per file in Table 1) using our tool move-r-patterns. The tool, generated patterns and scripts for replicating our experiments are included in our GitHub repository. We chose the patterns in the first set s.t. \( \overline{\text{occ}} \approx m \). This implies that when locating those patterns, we measure a blend of LF-, \( \Phi \)- and \( \text{rank-select} \) queries on \( L' \), since we perform \( \overline{\text{occ}} \approx m \Phi \) queries, \( 2m \) LF queries and at most \( 2m \) \( \text{rank-select} \) queries on \( L' \). The patterns in the second set were chosen s.t. \( \overline{\text{occ}} \approx 10^5m \). When locating those patterns, we practically only measure \( \Phi \) query performance. To measure count performance, we used the first set of patterns.

Figure 6 shows query performance versus index size. Out of the indexes without locate support, different block-rlbwt indexes provide the best trade-off between query performance and index size, depending on the repetitiveness and alphabet size of the text (see [8] for a more detailed discussion). \( \text{r-index-f} \) is consistently the smallest index, but also achieves low query throughput. Out of the dynamic indexes, online-rlbwt clearly performs better, because \text{rcomp-glfig} is consistently 3–4 times as large as it and often achieves a lower query throughput. \( \text{r-index} \) usually provides query performance similar to online-rlbwt while being only 1/3 as large. move-r outperforms \( \text{r-index} \) by factors between 3 and 35 (typically 15) while being 2–2.5 times as large. Comparing move-r with the fastest block-rlbwt index, respectively, move-r is 2–3 (typically 2) times as fast and 0.8–6 (typically 2) times as large. However, move-r can also be constructed without locate support, which roughly halves its size. This lessens (by a factor of 2) and sometimes cancels out the index size advantage of the respective fastest block-rlbwt index.

6 Conclusion

Overall, we have shown that the move data structure speeds up the r-index while causing an acceptable space increase and can be constructed efficiently. Regarding query throughput, Move-r outperforms the fastest other r-index supporting locate queries ([11] or [2]) by factors between 2 and 35 (typically 15) while being 0.8–2.5 (typically 2) times as large. Move-r can be constructed 0.9–2 (typically 2) times as fast while consuming 1/3–1 (typically 1/2) times as much memory. Compared with the fastest r-index supporting only count queries, Move-r achieves 2–3 (typically 2) times better query throughput while being 0.4–3 (typically 1) times as large.

References


**A Proof of Lemma 14**

**Lemma 14.** For $i \in [1, m]$, $p = SA^{-1}[z(y(i)) + 1]$ is the end position of a BWT run.

**Proof.** Let $p' = SA^{-1}[z(y(i) + 1)]$. By the definition of $P_{y(i)}$, $T_{z(y(i)) + 1} - 1$ does not start with $P_{y(i)}$, hence $L[p'] = T[z(y(i) + 1) - 1] \neq P[y(i)]$. The definition of $T_{z(y(i)) + 1}$ implies $T_{z(y(i)) + 1} < T_{z(y(i)) + 1}$ and therefore $p < p'$. Now suppose $p$ was not the end position of a BWT run. Since $L[p + 1] = L[p] = P[y(i)]$ implies $p + 1 < p'$, $T_{SA[p+1]}$ starts with
$P_{y(i)+1}$ and $T_{SA[LF(p+1)]}$ starts with $P_{y(i)}$. However, because $LF(p) < LF(p+1)$ follows with the LF-property, $T_{SA[LF(p+1)]}$ is lexicographically larger than $T_{i(y(i))}$, which contradicts the definition of $T_{i(y(i))}$, hence the claim is correct.

\section*{B More practical optimizations}

\subsection*{B.1 Practically optimized rank-select data structure}

Nishimoto and Tabei proposed the following data structures to implement $RS_L$:  
- $RS_{map}[1..\sigma]$: deterministic dictionary [21] storing the order-preserving mapping function from $\Sigma$ to the effective alphabet $\Sigma'$ of $\Sigma$  
- $RS_{rank}$: wavelet tree [3] of $L'$  
- $RS_{select}$ of size $r'$, where $RS_{select}[c][i] = select(L', c, i)$ for $c \in \Sigma'$ and $i \in [1, |Occ(c, L')|]$

With $RS_{rank}$, we can answer rank queries on $L'$ in $O(\log \log_\omega \sigma)$ time. With $RS_{map}$ and $RS_{select}$, we can answer select queries on $L'$ in $O(1)$ time. With the move data structure $M_{LF}$, the asymptotic runtime of the count algorithm is bounded by the runtime $O(m \log \log_\omega \sigma)$ (for $a = O(1)$) of the rank queries on the wavelet-tree of $L'$. This is also the case in the locate algorithm, if $occ = O(m \log \log_\omega \sigma)$ for $a = O(1)$.

We instead store $\sigma$ bit-vectors of size $r'$ each in $RS_{L'1[1..\sigma][1..r']}$, where $RS_{L'[c][i]} = 1 \iff L'[i] = c$. Then we can answer rank($L', c, i$) by computing $RS_{L'1[c].rank1}(i)$. Similarly, we compute $RS_{L'1[c].select1}(i)$ to answer select($L', c, i$).

If a character $c \in \Sigma$ occurs at least $r'/10$ times in $L'$, we implement $RS_{L'[1..r']}$ with an uncompressed bit vector and augment it with $O(1)$ time and $o(r')$ space rank$_1$ [23] and select$_1$ [7] data structures. Else, we use an array [20] to implement $RS_{L'[c][1..r]}$. This results in $O(\log \frac{r'}{n_c} + \log^4 n_c / \log r')$ time to answer a rank query on $L'$ and $O(\log^4 n_c / \log r')$ time to answer a select query on $L'$, where $c \in \Sigma$ and $n_c$ denotes the number of occurrences of $c$ in $L'$. Overall, we need $O(\sum_{i=1}^m (a + \log \frac{r'}{n_P[i]} + \log^4 n_P[i] / \log r')) = O(m(a + \log^3 r))$ time to answer a count query, because $1 \leq n_c < r' / c \in \Sigma$ and $r' = O(r)$, where $n_P[i]$ denotes the number of occurrences of $P[i]$ in $L'$. This is asymptotically worse than the $O(m(a + \log \log_\omega \sigma))$ time that we get when using a wavelet-tree of $L'$. In practice, our rank-select data structure reduces the running time of count- and locate queries (for $occ \lesssim m$) by a factor up to 3.

Since there are at most 9 characters $c \in \Sigma$ occurring at least $r'/10$ times in $L'$ (because $\$ occurs exactly once in $L'$), there are at most 9 uncompressed bit vectors in $RS_{L'}$, which need $9r' + o(r')$ bits. The size of the arrays in $RS_{L'}$ is bounded by $\sum_{c \in \Sigma} (n_c \log \frac{r'}{n_c} + 2n_c + o(n_c)) = \sum_{c \in \Sigma} (n_c \log \frac{r'}{n_c}) + 2r' + o(r') = r'\mathcal{H}_0(L') + 2r' + o(r')$ bits, which is asymptotically not larger than a huffman-shaped wavelet-tree of $L'$, which needs $r'\mathcal{H}_0(L') + o(r'\mathcal{H}_0(L') + 1) + \sigma \omega$ bits (see [16]), where $\mathcal{H}_0(L')$ is the Zeroth Order Entropy of $L'$. In practice, $RS_{L'}$ is roughly 2 times as large as a huffman-shaped wavelet-tree of $L'$.

\subsection*{B.2 Reducing index construction time in practice}

In [19], they showed how to construct $M_{idx}$ in $O(k' \log k')$ time using a binary search over $M_{idx}[1..k']$ for each entry in $M_{idx}[1..k']$. To speed up the construction in practice, we compute $\pi$ for $B_c(I)$ and use it to iterate over the output intervals of $B_c(I)$ in $O(k')$ time. We simultaneously iterate over the input intervals to keep track of the index $i$ of the input interval containing the starting position of the current ($j$-th) output interval. Hence, we can write $M_{idx}[\pi[j]] \leftarrow i$ for each $j$-th output interval. Since the construction of $\pi$ takes $O(k' \log k')$ time, this method still results in the same theoretical time to construct $M_{idx}$, but is much faster in practice.
We can apply a similar method to build $\text{SA}_\Phi$. We at first build the array $\text{SA}_\Phi'[1..r']$, where $\text{SA}_\Phi'[i] = \text{SA}[M_\Phi'[i+1] - 1]$, if $p_i$ is the end position of a BWT run and $\text{SA}_\Phi'[i] = \infty$, else. Then, we build the permutation $\pi'[1..r']$ storing the order of the values in $\text{SA}_\Phi'$ in $O(r' \log r')$ time. To iterate over the output intervals of $M_\Phi'$, we use the following tricks. Instead of splitting each input interval considered (and its corresponding output interval) at the offset $\text{offs}$, we split the input intervals before performing the balancing algorithm, because splitting an input interval can make the output interval containing the split $\in \{0, 1\}$, i.e., we can build $\text{SA}_\Phi'$ by simultaneously iterating over $M_\Phi'[\pi[1]], M_\Phi'[\pi[2]], ... , M_\Phi'[\pi[r']]$ and $\text{SA}_\Phi'[\pi'[1]], \text{SA}_\Phi'[\pi'[2]], ... , \text{SA}_\Phi'[\pi'[r']]$ in $O(r + r') = O(r)$ time and setting $\text{SA}_\Phi'[\pi'[i]] \leftarrow \perp$ for $i \in (r, r']$ in $O(r' - r) = O(r)$ time.

### B.3 Reducing index size in practice

We reduced the size of the move data structure by using the following tricks. Instead of storing $M_q$, we only store the offset of each output interval starting position in the input interval containing it, i.e, we store the arrays $M_p[1..k]$, $M_{offs}[1..k]$ and $M_{idx}[1..k]$, where $M_{offs}[i] = q_i - M_p[M_{idx}[i]]$, for $i \in [1, k]$. This allows us to compute $q_i$ with one random access to $M_p$ at the position $M_{idx}[i]$, which is irrelevant when evaluating a move query, because when computing move($M, i, x) = (i', x')$, we start scanning over $M_p$ at the position $M_{idx}[x]$ to find $x'$.

The length of the longest input interval is an upper bound of any value in $M_{offs}$. This implies that we can store $M_{offs}$ with the word width $\omega_{offs}$ by limiting the length of each input interval to $2^{\omega_{offs}}$. We can do this by iterating over the input intervals from left to right and splitting each input interval considered (and its corresponding output interval) at the offset $2^{\omega_{offs}}$ if it is longer than $2^{\omega_{offs}}$. We split the input intervals before performing the balancing algorithm, because splitting an input interval can make the output interval containing the split position $\omega$-heavy. Since we perform $\leq \frac{n}{2^{\omega_{offs}}}$ splits, the number of input- and output intervals increases by a factor $\leq 1 + \frac{n}{2^{\omega_{offs}}}$. We choose $\omega_{offs} = \min\{\omega \in \{8, 16, 24, 32, 40\} : \frac{n}{2^{\omega}} \leq \epsilon\}$ to bound the overall size of the resulting move data structure to $O((1 + \epsilon)\frac{n}{\omega_{offs}} k)$, for some $\epsilon > 0$. This suffices, if we assume $n \leq 2^{40}$, that is if the text file is smaller than $\approx 1$TB. In practice, $\epsilon = \frac{1}{5}$ turns out to be reasonable trade-off.

We also store $M_p$ and $M_{idx}$ with the minimum possible word widths $\omega_p = \min\{\omega \in \{8, 16, 24, 32, 40\} : n \leq 2^{\omega}\}$ and $\omega_{idx} = \min\{\omega \in \{8, 16, 24, 32, 40\} : k' \leq 2^{\omega}\}$, respectively. Finally, we store $M_p$, $M_{offs}$ and $M_{idx}$ interleaved with each other, to reduce the number of cache misses when performing move queries.

To further reduce the number of cache misses, we store $L'$ interleaved with the arrays of $M_{LF}$. Since $\text{SA}_\Phi'[i] \in [1, r'']$, we can store $\text{SA}_\Phi'$ with the same word width $\omega_{idx}$ as $M_{idx}'$. 
Engineering Zuffix Arrays

Paolo Boldi
Università degli Studi di Milano, Italy

Stefano Marchini
Università degli Studi di Milano, Italy

Sebastiano Vigna
Università degli Studi di Milano, Italy

Abstract
Searching patterns in long strings is a classical algorithmic problem with countless practical applications. Suffix trees and suffix arrays (and their variants) are a long-established solution that yields linear-time search (in the size of the pattern). In [5] it is shown that a z-map gadget can be attached to (enhanced) suffix arrays to improve their theoretical query time, obtaining a data structure called zuffix array. The main contribution of this paper is to show that a carefully engineered implementation of the z-map gadget does provide significant speedups with respect to enhanced suffix arrays on real-world datasets, albeit doubling the required space. In particular, for large alphabets we observe a sevenfold improvement in query time with respect to enhanced suffix arrays; even in the worst case (small alphabets), the query time is almost halved. Thus, zuffix arrays provide a very interesting new point in the space-time tradeoff spectrum.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Suffix trees, suffix arrays, z-fast tries

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.2

Supplementary Material Software (Source Code): https://github.com/smarchini/zuffix/tree/reproduce

1 Introduction

The suffix array [19] of a string \( s \) over an ordered alphabet \( \Sigma \) of \( \sigma \) characters is the array of the starting points of the suffixes of \( sS \) (where \( S \) is a character larger than any element of \( \Sigma \)), sorted lexicographically by the corresponding suffix. Suffix arrays are an extremely effective way of looking for all the occurrences of a pattern in a string, as once they are built (with some additional ancillary data) search requires only an amount of work linear in the length of the search pattern. A large body of research has gone into building and representing suffix arrays efficiently (e.g., in compressed form) [14, 8, 20, 18, 16, 22, 23, 10, 21, 6].

Interestingly, at the price of two additional (and very compressible) arrays of integers a suffix array can be used to represent (and navigate) implicitly the suffix tree associated with the string \( s \) [1]. Suffix trees are a special type of trie [17], which naturally suggests the possibility to enhance searches by using a z-map [2], obtaining what was called a zuffix tree in [5]. With this extra gadget in place, given a pattern \( p \) of length \( m \), after a preprocessing requiring time \( O(1 + (m \log \sigma)/w) \) (\( w \) being the machine word size), searches can be performed in \( O(\log m) \) search steps, each accessing a constant amount of information. The result is obtained only with high probability, and requires a verification phase (accessing \( m \))
characters), but the interesting fact is that the characters are accessed in sequential fashion at less than $\sigma$ different positions of the original text $s$. In other words, the described technique makes it possible to find with high probability the occurrences of pattern $p$ in $s$ using time $O((m \log \sigma)/w + \log m + \sigma)$ and $O((m \log \sigma)/B + \log m + \sigma)$ I/Os in the cache-oblivious model.

Whether (and under which conditions) this theoretical advantage in the cache-oblivious model reflects an improvement in actual searches is the question we aim to answer in this paper. We start from an enhanced suffix array, and we add a carefully engineered implementation of the z-map gadget; our experiments show that the resulting data structure does provide significant speedups with respect to enhanced suffix arrays on real-world datasets, albeit doubling the required space. It is worth noting that a z-map can be attached to any trie-based data structure, so this paper should be seen as a first effort in the direction of improving different types of structures based on suffix trees and suffix arrays.

All code used in our experiments, with full reproducibility instructions, is available at https://github.com/smarchini/zuffix/tree/reproduce.

## 2 Notation and Tools

### 2.1 Notation

Let $\Sigma$ be a fixed alphabet (of cardinality $\sigma$) not including the special symbol $\$$, and define $\hat{\Sigma} = \Sigma \cup \{\$\}$. The alphabet $\Sigma$ comes endowed with a specified (arbitrary) total order, that is inherited by $\hat{\Sigma}$ with the proviso that $\$$ is larger than any other character. We use $\leq$ to denote the induced lexicographic order on $\hat{\Sigma}^*$, whereas $\preceq$ is used to denote the prefix order.

If $x \in \hat{\Sigma}^*$ is a string, $x$ juxtaposed with an interval is the substring of $x$ with those indices (indices start from 0). Thus, for instance, $x[a..b]$ is the substring of $x$ starting at position $a$ (inclusive) and ending at position $b$ (inclusive). We will write $x[a]$ for $x[a..a]$ and $x[a.]$ for $x[a..|x| - 1]$. By definition, $x[|x|..] = \varepsilon$.

### 2.2 Machine model

We analyze our algorithms on a unit-cost word RAM with word size $w$ in the cache-oblivious model [11]. In this model, the machine has a two-level memory hierarchy, where the fast level has an unknown size of $M$ words and the slow level has an unbounded size and is where our data reside. We assume that the fast level plays the role of a cache for the slow level with an optimal replacement strategy where the transfers (a.k.a. I/Os) between the two levels are done in blocks of unknown size of $B \leq M$ words; the I/O cost of an algorithm is the total number of such block transfers. Scanning is a fundamental building block in the design of cache-oblivious algorithms: given an array of $N$ contiguous items the I/Os required for scanning it completely is $O(1 + N/B)$.

### 2.3 Tries

A (compacted) trie [17] over $\Sigma$ is a rooted tree such that

- every node $\alpha$ is endowed with a string $c_{\alpha} \in \Sigma^*$ (the compacted path of $\alpha$);
- every arc connecting an internal node $\alpha$ with one of its children $\alpha'$ is labelled with a character $c_{\alpha,\alpha'} \in \Sigma$ and $c_{\alpha,\alpha'} \neq c_{\alpha,\alpha''}$ for any two distinct children $\alpha'$ and $\alpha''$ of $\alpha$;
- every internal node has at least two children.
For every node $\alpha$ of a compacted trie, we define its name $n_\alpha \in \Sigma^*$ and its extent $e_\alpha \in \Sigma^*$ recursively as follows:

- $n_{\text{root}} = \varepsilon$;
- $e_\alpha = n_\alpha c_\alpha$;
- if $\alpha'$ is a child of $\alpha$, then $n_{\alpha'} = e_\alpha c_\alpha,\alpha'$.

For any given finite nonempty prefix-free language $L \subseteq \Sigma^*$, the compacted trie of $L$ is the only compacted trie $T(L)$ over $\Sigma$ such that $L$ is the set of all the extents of the leaves of $T(L)$.

In Figure 1, we show an example of a binary trie with the nomenclature just introduced (and some more that will be introduced below).

![Figure 1](image.png)

**Figure 1** (above) The compacted trie $T(L)$ (with the corresponding nomenclature) and its z-map. Here $\Sigma = \{0, 1\}$ and $L = \{001001010, 0010011010010, 0010011010\}$.

Given a compacted trie over $\Sigma$, and given a string $p$, we let $\text{exit}(p)$ (the exit node of $p$) be the node having a name that is the longest possible prefix of $p$. (See Figure 1 for an example of an exit node).

Note that:

- Proposition 1 ([5]). Let $L \subseteq \Sigma^*$ be finite, nonempty, and prefix-free, and consider the trie $T(L)$. A string $p \in \Sigma^*$ is a prefix of some element of $L$ if and only if $p \preceq e_{\text{exit}(p)}$. Moreover, if the latter happens then the set

$$\{e_\alpha \mid \alpha \text{ is a leaf descendant of } \text{exit}(p)\}$$

is precisely the set of all $x \in L$ such that $p \preceq x$.

In the following, we shall always assume that $\Sigma$ is a fixed alphabet, $\sigma = |\Sigma|$ and that $L \subseteq \Sigma^*$ is finite, nonempty and prefix-free.

### 3. Z-fast tries

Let us assume that we have built the trie $T(L)$ for a given language $L$ of size $n$. Proposition 1 gives an easy way to determine if a string $p$ of length $m$ is a prefix of some element of $L$: it is enough to locate the node $\gamma = \text{exit}(p)$ and then to check whether $p$ is a prefix of the extent of $\gamma$ or not. Moreover, the second part of the statement suggests which elements of $L$ have $p$ as a prefix.
Locating \(\text{exit}(p)\) can be done trivially in \(O(m\sigma)\) steps, going down in the trie starting from the root. An alternative faster solution is that of enriching the compacted trie with a kind of acceleration map, the \textit{z-map} [2], which makes it possible navigate the trie using a number of accesses to the map logarithmic in the length of the search string.

### 3.1 Short strings

The idea comes in two flavors, depending on the length of the strings in the set \(L\). Let us start with the (simplest) case of “short” strings. The \textit{2-fattest number} of an interval \([a..b]\) of non-negative integers is the (unique) integer in \([a..b]\) that is divisible by the largest power of two. The \textit{handle} \(h_\alpha\) of a node \(\alpha\) of a trie is the prefix of \(e_\alpha\) whose length is the 2-fattest number in \([|n_\alpha|..|e_\alpha|]\) (the \textit{skip interval} of \(\alpha\)).

In Figure 1 we show the (length of the) handles of each node, including the leaves: the handle is the string starting from the root and ending just above the dashed lines you can see in each node.

\[\text{Definition 2 (z-map).}\]

The \textit{z-map} \(Z(-)\) for the trie \(T(L)\) is a map from elements of \(\Sigma^*\) to nodes in the trie, which maps \(h_\alpha\) to \(\alpha\) for each internal node \(\alpha\).

The z-map can be stored using any static dictionary with constant-time access; we assume that the dictionary returns the special value \(\perp\) whenever it is queried with a key that is not in the dictionary.

The usefulness of the z-map is made evident by the following result:

\[\text{Theorem 3 ([5]).}\]

If the length of the strings in \(L\) is bounded by \(O(w/\log \sigma)\), using the z-map for the trie \(T(L)\) it is possible to establish if a pattern \(p \in \Sigma^*\) of length \(m\) is a prefix of some string of \(L\) in \(O(\log m + \sigma)\) time and I/0s.

The algorithm, presented in [5], performs a sort of a binary search (called \textit{fat binary search}), with at most one access to the actual trie structure (to enumerate the children of a node). Moreover, the comparisons between \(p\) and other strings require at most \(\sigma\) scans of overall \(m\) characters.

### 3.2 Long strings

If the length of the strings in \(L\) is not bounded by \(O(w/\log \sigma)\), the map \(Z(-)\) described above uses superlinear space and superconstant time at every access. This is why the “long string” version of dynamic z-fast tries [3] replaces handles with \textit{signatures}: in the z-map, instead of storing pairs \((h_\alpha, \alpha)\) we store pairs \((H(h_\alpha), \alpha)\) where \(H(-)\) is a suitably chosen signature hash function. The signature-based version is designed to work with sets of at most \(2^{O(w)}\) strings of length up to \(2^{O(w)}\), but this time the fat binary search returns the correct result only with high probability.

If we use signatures of size \((c + \varepsilon) \log n\), with \(c \geq 2\), we will find distinct hash values for all handles after a constant expected number of attempts. Indeed, under a full randomness assumption, the probability of having a hash collision between distinct handles is at most

\[1 - e^{-n^2/2^c(c+\varepsilon) \log n} = 1 - e^{-n^2/2n^c\varepsilon} \leq \frac{1}{2n^{c-2+\varepsilon}} \to 0\]

as \(n \to \infty\). Once we are sure that the signatures are all distinct, in estimating the probability of error of a fat binary search we have to care just about at most \(\log m\) false-positive results in queries performed during the fat binary search, which by the union bound happens with probability at most

\[2^{-(c+\varepsilon) \log n} \log m = \frac{1}{n^c} O(w) = o\left(\frac{1}{n^c}\right)\]
Note that each time we have to query the z-map, we must compute the hash of a potentially long prefix: for this purpose, the dynamic z-fast trie uses hash functions that can hash any prefix of the pattern $p$ in constant time after preprocessing $p$ in time $O(1 + (m \log \sigma)/w)$ and storing a linear amount of information (see Section 5).

Thus, for the signature-based case, we have:

- **Theorem 4 ([5])**. Under the full randomness assumption, let $c \geq 2$ and assume that $Z(-)$ stores $((c+\varepsilon) \log n)$-bit hash values without collisions. Then, in time $O((m \log \sigma)/w + \log m + \sigma)$ and with $O((m \log \sigma)/B + \log m + \sigma)$ I/Os it is possible to establish if a pattern $p \in \Sigma^*$ of length $m$ is prefix of some string of $L$; the result will be correct with probability at least $1 - o(1/n^c)$; when the algorithm detects that an error has occurred, the correct result is obtained by resorting to the standard naive search on the trie, which requires $O(m\sigma)$ time and I/Os.

### 4 Suffix Trees, Suffix Arrays, and Zuffification

Given a string $s \in \Sigma^*$, the suffix tree [24] $T(s)$ is the trie over the alphabet $\hat{\Sigma}$ containing all suffixes of $s$. Although one can build in linear time the suffix tree of a string, there are approaches based on suffix arrays [19] that are much more efficient in practice. The suffix array $sa$ of $s$ is the permutation of $\{0, 1, \ldots, n\}$ such that

$$s[sa[0] \ldots] < s[sa[1] \ldots] < \cdots < s[sa[n] \ldots].$$

Not only there are linear-time algorithms to build the suffix array: at the price two additional (and very compressible) arrays of integers, the suffix array can be made into a so-called enhanced suffix array, and then used to represent and navigate implicitly the suffix tree [1].

The z-map gadget described in Section 3 is applicable to all prefix-free languages, so in particular it can be used for suffix trees, and since the enhanced suffix array is equivalent to the suffix tree, it can be used to build the z-map over the suffix tree/array: this operation is called zuffification of the suffix tree/array [5].

To build the z-map, we perform a depth-first visit of the suffix tree: in fact, we will simulate it on a suffix array. For each internal node, we can compute its handle (as explained above), the corresponding hash, and store in a static dictionary the correspondence between the hash and the node; hash collisions will cause some correspondence to be wrong, but we will handle this problem as part of the resilience of fat binary search to hash collisions. Computing a hash for a string $x$ needs time $O(1 + (|x| \log \sigma)/w)$, which would imply quadratic construction and preprocessing time: in the next sections, we are going to discuss how to reduce it to linear time.

### 5 Constant Time Prefix Hashing

We start by studying how to compute the hash of a prefix of a string $s$ in constant time. Note that in theory this is not enough to obtain linear construction time, but we will see that in practice this kind of approach works very well.

When looking for the exit node of a string we query the z-map on prefixes of the pattern. We will use a scheme that scans the entire string only once and enables computing the signature of any prefix in constant time. This scheme exploits the pause&resume technique, exposed by several general-purpose hash functions: to compute the signature of large files or streaming data it is common practice to process the sequence in chunks; in other words, we read the first portion (i.e., a chunk) of the string, compute an intermediate internal state of the hash function, and continue from there without keeping the whole string in memory.
Consider a hash function \( H \) offering the \texttt{pause\&resume} capability, and fix a constant block-size \( B \): during the computation of \( H(s) \), keep track of the internal state of the function (in a suitable \texttt{internal-states} table) at \( H(s[\ldots B]) \), \( H(s[\ldots 2B]) \), \( H(s[\ldots 3B]) \), \ldots, and so on. The signature of any prefix \( s[\ldots k] \) can then be computed by loading the nearest internal state, \( H(s[\ldots [k/B]]) \), and resuming the hash function from there, reading at most \( B − 1 \) additional characters of \( s \).

With this observation in mind, we can try out some of the fastest general-purpose hash functions available from off-the-shelf libraries to trade off a (maybe only theoretically) slower construction time for practically faster search queries.

### 6 Constant Time Substring Hashing

If we want to obtain linear construction time, we need to be able to compute the hash of any substring of a string \( s \) in constant time. For this purpose, we can use hash functions based on Cyclic Redundancy Checks (CRCs). CRCs are a widespread family of hash functions based on the division in the ring of polynomials with coefficients in the finite field \( \mathbb{F}_2 \), the field of integers modulo 2. In this field, addition and subtraction correspond to bitwise-xor and multiplication is a “carry-less product”, which in binary arithmetic can be thought of as a repeated shift\&xor operation similar to long multiplication. Hence, CRCs can be implemented using either hardware facilities (e.g., the \texttt{pclmulqdq} assembly instruction, i.e., the carry-less product of two 64-bit values) or with bitwise operations.

In the computation of CRCs, the whole string is interpreted as a long binary number whose digits become the coefficients of a polynomial; the signature is therefore the remainder of the polynomial modulo a fixed irreducible polynomial, the \textit{generator polynomial}. More precisely, CRC hash functions are defined as:

\[
H(s) = s(x)x^k \mod g(x) \iff s(x)x^k = q(x)g(x) + H(s)
\]

where \( s(x) \) is the binary representation of the string \( s \) thought of as a polynomial in the variable \( x \), \( q(x) \) is the quotient polynomial, and \( g(x) \) is the generator polynomial. The degree \( k \) of the generator polynomial \( g(x) \) determines the size of the signature.

As part of the SSE4.2 extensions, modern CPUs provide full hardware support (i.e., using a single assembly instruction) for computing the 32-bit CRC used in the standard iSCSI; this CRC adopts a generator known as the Castagnoli polynomial \[7\].

CRCs are linear functions, and their internal state corresponds to the signature of the string up to that position. Given two strings \( a \) and \( b \), we show how to compute efficiently \( H(ab) \) starting from \( H(a) \) and \( H(b) \). In particular, we discuss how to append (internally) and prepend zero-bit strings to a signature; then, we can leverage the linearity properties to compose and decompose signatures (\( \oplus \) denotes bitwise-xor):

\[
H(ab) = H(ab[0] \oplus 0^{|a|b}) = H(ab[0]) \oplus H(0^{|a|b}).
\]

We start by noting that \( H(0^\ell s) = H(s) \) for any string \( s \) and for any integer \( \ell \): adding high-degree zero-coefficient terms to a polynomial does not affect its value. Appending zeroes, instead, consists in a polynomial multiplication between \( s(x) \) (i.e., the polynomial of \( s \)) and \( x^\ell \). To compute \( H(s0^\ell) \) we use the fact that internal states are signatures of prefixes. Instead of employing the shift\&xor method, this operation is typically performed employing a lookup table for the linear transformation of appending power-of-two many zeroes to an internal state: by querying the table for every 1-bit in the binary representation of \( \ell \) we can simulate encountering \( \ell \) consecutive zeroes after reading the string.
Constant-time substring hashing is possible after precomputing the internal-state table as we discussed in Section 5, and a similar table for strings of the form $10^\ell$. Given two integers $i$ and $j$ such that $i < j$, the internal-state table of the string $s$ enables us to compute $H(s[\ldots i])$ and $H(s[\ldots j])$ in constant time; then, we can apply the linear transformation that appends $j - i$ zero-symbols to $H(s[\ldots i])$ and use linearity to calculate $H(s[i + 1 \ldots j])$ in constant time. However, we found that in practice the second table can be avoided since the cost of computing its values on-the-fly is logarithmic in $\ell$.

7 Collision Detection during Fat Binary Search

As we already mentioned, we query the z-map with hashes of prefixes. It is thus possible that a prefix has the same hash value as a handle, even if it is a different string. If this happens, the exit node we obtain as a result of the fat binary search will be incorrect. To detect this behavior, we need to check after the search whether the node we found is really the exit node.

The standard `memcmp` approach consists of performing a full string comparison between the extent of the exit node and the pattern. This check cannot fail to detect an error.

An alternative, the `signature approach`, works as follows: instead of comparing strings character by character, in the case of CRC hashing we use constant time substring hashing to compare the pattern with the original text. The rationale is that since we can hash the original text and the pattern in constant time this check, albeit not exact, might be faster than the `memcmp` approach, and might provide an interesting tradeoff.

8 Experimental Evaluation

We ran our benchmarks on a 12th Gen Intel® Core™ i7-12700KF (Alder Lake) workstation, with 64 GiB of DDR4 RAM, GNU/Linux 6.1.14, and the GNU C Compiler (GCC) 13.0.1. We compiled the external dependencies (most notably, the hash function implementations) with optimizations enabled; note, in particular, that our architecture supports the BMI and the AVX2 instruction set. We tested two distinct CRC implementations using two different 32 bit generator polynomials. In particular, we used the software-based CRC implementation present in Zlib [12] and the hardware-based implementation in Facebook’s Folly\(^1\). The hardware-based implementation uses the `crc` and the `pclmulqdq` assembly instructions, with the optimizations discussed in [15].

The tests were performed on datasets from the standard Pizza & Chilli Corpus [9], which includes various types of text that have been selected as a representative sample of different applications of indexed text search. To account for the edge cases, we also included artificially generated texts made of random strings varying in alphabet size, and the highly repetitive Fibonacci words [4]. Each character in the randomly generated texts is extracted with equal probability from a set of 4 (`rand-4`) and 62 (`rand-62`) ASCII symbols. These random texts are used to reproduce some of the properties in the `dna` and in the `english` dataset in Pizza & Chilli. We report the data of the most important findings and present some more general considerations based on the insights given by the full set of experiments.

\(^1\) https://github.com/facebook/folly/
8.1 Variants

We consider a number of implementations described in Table 1. Our baseline is enhanced, our implementation of the enhanced suffix array. The other variants are combinations of hash functions, either with memcmp or signature string comparisons.

Table 1: Naming conventions explained. $B$ is the block size, a parameter that determines the size of a chunk in the internal-state table. $S$ is the size (in bits) of the internal state of the hash function. For XXH3 we used the functions exposed by the library to save and load the internal state, which is suboptimal for our use case.

<table>
<thead>
<tr>
<th>$B$</th>
<th>$S$</th>
<th>variant</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>—</td>
<td>enhanced</td>
<td>Abouelhoda <em>et al.</em>’s enhanced suffix array [1]</td>
</tr>
<tr>
<td>480 B</td>
<td>192</td>
<td>memcmp-wyhash</td>
<td>suffix array with string comparison and wyhash</td>
</tr>
<tr>
<td>64 kB</td>
<td>4608</td>
<td>memcmp-xxh3</td>
<td>suffix array with string comparison and XXH3</td>
</tr>
<tr>
<td>512 B</td>
<td>32</td>
<td>memcmp-folly</td>
<td>suffix array with string comparison and hardware CRC</td>
</tr>
<tr>
<td>1 kB</td>
<td>32</td>
<td>memcmp-zlib</td>
<td>suffix array with string comparison and software CRC</td>
</tr>
<tr>
<td>512 B</td>
<td>32</td>
<td>signature-folly</td>
<td>suffix array with signature comparison and hardware CRC</td>
</tr>
<tr>
<td>1 kB</td>
<td>32</td>
<td>signature-zlib</td>
<td>suffix array with signature comparison and software CRC</td>
</tr>
</tbody>
</table>

8.2 Statistics

In Table 2, we collected a series of statistics for each text. These statistics are meant to integrate those previously collected by the authors of the Pizza&Chili Corpus. Our additional data focus on suffix-tree structural properties and hash-function qualities: as we will see, our additional data make it possible to give a detailed explanation of the results of our experiments.

8.3 Suffix tree

By reporting the depth of the suffix tree, counting the number of internal nodes, and measuring the average length of name, handle, and extent of the internal nodes we aim to identify some of the variables needed to take advantage of the zuffification. In particular, we remark that the $z$-map speeds up the top-down navigation of suffix trees. The enhanced suffix array implements the suffix tree navigation in such a way that it induces a first-child next-sibling top-down tree traversal. Therefore, the enhanced suffix array can efficiently walk from the root to the exit when the alphabet is small, but we expect it to be slower on large alphabet texts. The running time of the fat binary search, instead, is not as directly dependent on the size of the alphabet.

8.4 Hash functions

We implemented the $z$-map using an open-addressing hash table with linear probing. For each hash function, we display the ratio number of distinct handles with equal signatures (*store collisions*). Note that with signatures of 64 bits or more we find no collisions. Moreover, collision problems can be partially mitigated with a few inexpensive runtime checks, preceding the techniques discussed in section 7. In a single search query, the $z$-map is expected to find nodes that lay on the path from the root to the exit node, whose handle length is the 2-fattest number in its skip interval. With the implicit representation of the tree provided
by the enhanced suffix array, ancestor queries are trivially answered in constant time by comparing lcp intervals [1]. This consideration will intuitively lead us to the conclusion that collisions will not play the biggest role in the choice of hash functions, an intuition that will be confirmed by the experimental results.

8.5 Space

The number of internal nodes also counts how many handles are stored in the z-map and it can be used to estimate the space consumed by the data structure. Consider a standard (non-compressed) implementation of the enhanced suffix array. Given a text of length $n$, let $w \geq \lceil \lg n \rceil$ be the size in bits of an unsigned integer large enough to hold indices of the text, and let $B$ be the block size of a hash function with an internal state of size $S$. We denote with $\text{slack}(\# \text{ internal nodes})$ the function that determines the size of the hash table given the number of internal nodes; in our implementation, tables are sized as powers of two, and they are never filled more than $2/3$. The space used to implement a suffix array is thus given by the sum of the space occupied by the enhanced suffix array (suffix array, lcp array, additional array),

\[
\text{space}(sa) = nw \\
\text{space}(lcp) = (n + 2)w \\
\text{space}(add) = nw,
\]

by the space occupied by the z-map (hash and node),

\[
\text{space}(Z) = (2w + (c + \varepsilon) \log n) \cdot \text{slack(\# internal nodes)}
\]

and by the space occupied by the internal-state table of the hash function in the signature case

\[
\text{space}(H) = nS/B.
\]

The block size and the state size used in our implementation are reported in Table 1 and, in our benchmarks, we set $w$ to 64. While performing string matching, the suffix array consumes additional $mS/B$ space for the internal-state table of the pattern. We tuned the block size of the hash function for speed and we obtained internal-state tables up to 32 times smaller than the strings.

8.6 Performance

A well-known assumption in stringology is that the pattern is orders of magnitude shorter than the text. From a theoretical standpoint, this folklore assumption can be justified by analyzing the suffix tree of the text, as we did in Table 2. In a suffix tree, the string-matching problem is, fundamentally, an exit node search followed by a string comparison. The names of the leaves provide a good description of the longest “meaningful” patterns we may search for: all the strings that are longer than the maximum name have zero or only one occurrence in the text. If two strings are longer than the maximum name in the suffix tree and if one of them is the prefix of the other, then the running time of the string-matching algorithm differs by only a (longer) string comparison. By considering only the patterns whose length is smaller than the maximum name length, we aim to focus only on the most significant portion of the performance plot.

The texts from the Pizza&Chilli Corpus come from real sources, and they are meant to be representative of real-life applications of text indexing. The two random texts and the
Engineering Zuffix Arrays

Table 2 Statistics for each text dataset.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>DNA</th>
<th>Proteins</th>
<th>dblp-xml</th>
<th>Sources</th>
<th>English</th>
<th>Rand-4</th>
<th>Rand-62</th>
<th>Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td>σ</td>
<td>4</td>
<td>27</td>
<td>97</td>
<td>239</td>
<td>239</td>
<td>4</td>
<td>62</td>
<td>2</td>
</tr>
<tr>
<td>size</td>
<td>200 MiB</td>
<td>200 MiB</td>
<td>200 MiB</td>
<td>200 MiB</td>
<td>200 MiB</td>
<td>200 MiB</td>
<td>200 MiB</td>
<td>200 MiB</td>
</tr>
<tr>
<td>suffix tree depth</td>
<td>165</td>
<td>391</td>
<td>123</td>
<td>3237</td>
<td>122</td>
<td>65</td>
<td>48</td>
<td>2</td>
</tr>
<tr>
<td># internal nodes</td>
<td>142M</td>
<td>268M</td>
<td>100M</td>
<td>131M</td>
<td>129M</td>
<td>10M</td>
<td>35M</td>
<td>9M</td>
</tr>
<tr>
<td>avg(# internal)</td>
<td>18</td>
<td>252</td>
<td>37</td>
<td>149</td>
<td>124</td>
<td>13</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>avg(# internal)</td>
<td>56</td>
<td>380</td>
<td>144</td>
<td>566</td>
<td>151</td>
<td>13</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>construction time</td>
<td>12.10s</td>
<td>17.34s</td>
<td>9.97s</td>
<td>10.85s</td>
<td>11.72s</td>
<td>12.15s</td>
<td>12.30s</td>
<td>280.94ms</td>
</tr>
<tr>
<td>suffix tree space</td>
<td>4.7 GiB</td>
<td>4.7 GiB</td>
<td>4.7 GiB</td>
<td>4.7 GiB</td>
<td>4.7 GiB</td>
<td>4.7 GiB</td>
<td>121.20 MiB</td>
<td></td>
</tr>
</tbody>
</table>

wyhash (64 bit pause&resume)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>DNA</th>
<th>Proteins</th>
<th>dblp-xml</th>
<th>Sources</th>
<th>English</th>
<th>Rand-4</th>
<th>Rand-62</th>
<th>Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td>zification</td>
<td>4.73s</td>
<td>6.60s</td>
<td>3.80s</td>
<td>5.76s</td>
<td>4.76s</td>
<td>4.23s</td>
<td>1.71s</td>
<td>8 m 24s</td>
</tr>
<tr>
<td>z-map space</td>
<td>5.02s</td>
<td>6.71s</td>
<td>4.39s</td>
<td>5.64s</td>
<td>34.8s</td>
<td>4.24s</td>
<td>1.60s</td>
<td>5 m 22s</td>
</tr>
<tr>
<td>store collisions</td>
<td>8 GiB</td>
<td>8 GiB</td>
<td>8 GiB</td>
<td>8 GiB</td>
<td>8 GiB</td>
<td>2 GiB</td>
<td>512 MiB</td>
<td></td>
</tr>
</tbody>
</table>

xxh3 (128 bit pause&resume)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>DNA</th>
<th>Proteins</th>
<th>dblp-xml</th>
<th>Sources</th>
<th>English</th>
<th>Rand-4</th>
<th>Rand-62</th>
<th>Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td>zification</td>
<td>14.2s</td>
<td>12.2s</td>
<td>10.1s</td>
<td>11.2s</td>
<td>12.7s</td>
<td>13.8s</td>
<td>1.80s</td>
<td>526 ms</td>
</tr>
<tr>
<td>z-map space</td>
<td>1.56%</td>
<td>1.65%</td>
<td>1.52%</td>
<td>1.50%</td>
<td>1.52%</td>
<td>0.22%</td>
<td>0.22%</td>
<td></td>
</tr>
<tr>
<td>store collisions</td>
<td>62.9s</td>
<td>55.0s</td>
<td>41.5s</td>
<td>53.0s</td>
<td>56.2s</td>
<td>58.4s</td>
<td>16.5s</td>
<td>4.58s</td>
</tr>
<tr>
<td>store collisions</td>
<td>1.71%</td>
<td>0.65%</td>
<td>1.16%</td>
<td>1.52%</td>
<td>1.50%</td>
<td>1.74%</td>
<td>0.22%</td>
<td></td>
</tr>
</tbody>
</table>

Fibonacci words dataset are artificial and they are best used as a means of comparison. Each text is trimmed to 200 MiB, so the results are normalized in size; the only exception is the text for the Fibonacci words, whose length is $F_{35}$. The symbols are always considered to be byte-based, so in the Pizza&Chilli Corpus we do not split English words into terms and the gene DNA nucleotides (or any unknown choice among them) are represented with their standard ASCII characters.

8.7 Construction

Albeit the general-purpose hash functions with the pause&resume capabilities discussed in Section 5 induce a suboptimal construction time, in practice the best functions available from off-the-shelf libraries are faster than CRCs, while also offering a wider output, leading to fewer collisions. In Table 2 we report the running time of zuffification. This result can be compared with the construction time of the enhanced suffix array, also reported in the table. For the construction of the suffix array we are using Ilya Grebnov’s libsais [13] compiled with OpenMP support. To the best of our knowledge, this is the fastest suffix array construction algorithm currently available. Note that our current implementation of zuffification is not multi-threaded, so the work done to construct the z-map is much smaller than the work required to build the enhanced suffix array. The size of the z-map is also reported in the table: it is comparable to the size of the enhanced suffix array for 32-bit CRCs, it becomes slightly larger in the case of 64-bit hashes, and it is almost twice as big for 128-bit hashes. The only exception is rand-64 because the number of internal nodes in that case is very small.

Fibonacci words contain, asymptotically, the maximum number of repetitions that may be contained in any string. Fibonacci words are a worst-case example for the construction algorithm and in the fibonacci column of Table 2 we can see how the theoretical results are reflected in practice: zuffification under the general-purpose pause&resume hash functions can
be several times slower than CRCs. Nevertheless, in all the other (more realistic) scenarios that we have studied, the general-purpose hash functions are competitive and sometimes outperform the theoretically faster hash functions offering constant-time substring hashing. This result is mostly a consequence of the statistical properties of the text and it is not just a matter of its size. The fibonacci corpus maximizes the length of the names; thus, zuffification can be significantly slower than in other corpora even if its text is much smaller.

Table 3 Raw performance (in nanoseconds) of string search by pattern size.

<table>
<thead>
<tr>
<th>size</th>
<th>enhanced</th>
<th>mem-wyhash</th>
<th>mem-xxh3</th>
<th>mem-folly</th>
<th>mem-zlib</th>
<th>sgn-folly</th>
<th>sgn-zlib</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>dna</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10^1</td>
<td></td>
<td>1505</td>
<td>720</td>
<td>1178</td>
<td>731</td>
<td>745</td>
<td>835</td>
</tr>
<tr>
<td>10^2</td>
<td></td>
<td>2009</td>
<td>1237</td>
<td>2063</td>
<td>139</td>
<td>1696</td>
<td>139</td>
</tr>
<tr>
<td>10^3</td>
<td></td>
<td>1996</td>
<td>1655</td>
<td>2643</td>
<td>1481</td>
<td>2740</td>
<td>1760</td>
</tr>
<tr>
<td>10^4</td>
<td></td>
<td>1998</td>
<td>2014</td>
<td>3269</td>
<td>1800</td>
<td>3497</td>
<td>2059</td>
</tr>
<tr>
<td></td>
<td></td>
<td>proteins</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10^1</td>
<td></td>
<td>3153</td>
<td>814</td>
<td>1365</td>
<td>848</td>
<td>973</td>
<td>1195</td>
</tr>
<tr>
<td>10^2</td>
<td></td>
<td>3267</td>
<td>1125</td>
<td>1710</td>
<td>1116</td>
<td>1637</td>
<td>1350</td>
</tr>
<tr>
<td>10^3</td>
<td></td>
<td>3326</td>
<td>1587</td>
<td>2345</td>
<td>1487</td>
<td>2772</td>
<td>1758</td>
</tr>
<tr>
<td>10^4</td>
<td></td>
<td>3323</td>
<td>2495</td>
<td>3872</td>
<td>2180</td>
<td>4851</td>
<td>2447</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dblp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10^1</td>
<td></td>
<td>4333</td>
<td>795</td>
<td>1059</td>
<td>811</td>
<td>887</td>
<td>1007</td>
</tr>
<tr>
<td>10^2</td>
<td></td>
<td>5815</td>
<td>1232</td>
<td>1708</td>
<td>1196</td>
<td>1955</td>
<td>1448</td>
</tr>
<tr>
<td>10^3</td>
<td></td>
<td>5736</td>
<td>1598</td>
<td>2281</td>
<td>1500</td>
<td>3037</td>
<td>1755</td>
</tr>
<tr>
<td>10^4</td>
<td></td>
<td>6810</td>
<td>850</td>
<td>1255</td>
<td>891</td>
<td>994</td>
<td>1103</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sources</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10^1</td>
<td></td>
<td>7300</td>
<td>1220</td>
<td>1760</td>
<td>1229</td>
<td>1884</td>
<td>1484</td>
</tr>
<tr>
<td>10^2</td>
<td></td>
<td>7379</td>
<td>1695</td>
<td>2382</td>
<td>1605</td>
<td>2959</td>
<td>1858</td>
</tr>
<tr>
<td>10^3</td>
<td></td>
<td>7352</td>
<td>2612</td>
<td>3915</td>
<td>2337</td>
<td>5068</td>
<td>2588</td>
</tr>
<tr>
<td>10^4</td>
<td></td>
<td>6137</td>
<td>799</td>
<td>1236</td>
<td>859</td>
<td>931</td>
<td>1029</td>
</tr>
<tr>
<td></td>
<td></td>
<td>english</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10^1</td>
<td></td>
<td>6365</td>
<td>1203</td>
<td>1888</td>
<td>1166</td>
<td>1760</td>
<td>1445</td>
</tr>
<tr>
<td>10^2</td>
<td></td>
<td>6424</td>
<td>1625</td>
<td>2437</td>
<td>1535</td>
<td>2819</td>
<td>1825</td>
</tr>
<tr>
<td>10^3</td>
<td></td>
<td>6458</td>
<td>2428</td>
<td>3735</td>
<td>2219</td>
<td>4843</td>
<td>2477</td>
</tr>
<tr>
<td>10^4</td>
<td></td>
<td>5033</td>
<td>716</td>
<td>1174</td>
<td>721</td>
<td>737</td>
<td>835</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rnd-4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10^1</td>
<td></td>
<td>5033</td>
<td>742</td>
<td>1228</td>
<td>723</td>
<td>809</td>
<td>916</td>
</tr>
<tr>
<td>10^2</td>
<td></td>
<td>5028</td>
<td>748</td>
<td>1205</td>
<td>717</td>
<td>821</td>
<td>918</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rnd-62</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10^1</td>
<td></td>
<td>160</td>
<td>163</td>
<td>186</td>
<td>161</td>
<td>174</td>
<td>236</td>
</tr>
<tr>
<td>10^2</td>
<td></td>
<td>212</td>
<td>217</td>
<td>273</td>
<td>207</td>
<td>508</td>
<td>285</td>
</tr>
<tr>
<td>10^3</td>
<td></td>
<td>325</td>
<td>536</td>
<td>712</td>
<td>570</td>
<td>1575</td>
<td>815</td>
</tr>
<tr>
<td>10^4</td>
<td></td>
<td>638</td>
<td>1060</td>
<td>1473</td>
<td>1083</td>
<td>3110</td>
<td>1213</td>
</tr>
</tbody>
</table>

8.8 Search queries

In Table 3 we report the query performance of all datasets for medium-length patterns, whereas in Figures 2–4 we show the performance of each variant under four significantly different scenarios and a wide range of lengths. Each plot reports the running time of search queries. Expectedly, the running time increases with the length of the pattern. For each pattern size, we generated 10,000 patterns extracted from random positions of the text and we measured the running time of each variant using standard benchmarking techniques. The running time reported in the plots is meant to be representative of that of a single search query; in particular, we avoided the potential caching speedups that may occur when several operations of the same kind happen in near succession. We remark that suffix arrays are expected to access the underlying suffix tree only once, and thus the underlying tree could even be stored in external memory, making the space occupation of the whole structure a less daunting obstacle than what might appear from the raw numbers in Table 2.
(a) dna: highly-repetitive small alphabet.

(b) dblp-xml: highly-repetitive large alphabet.

Figure 2 Raw performance of string search by pattern size.
(a) rand-4: barely repetitive small alphabet.

(b) rand-62: barely repetitive large alphabet.

Figure 3 Raw performance of string search by pattern size.
Figure 4 Raw performance of string search by pattern size.
The signature variants may return false positives: if there are any for some pattern, we report their number on the plot (the time required to check for false positives is included in our benchmarks). Counting the false positives in signature variants is also indicative of the performance of their memcmp counterpart: when the z-map fails to find the exit node, the string search algorithm (implicitly) falls back to a regular (non-z-fast, but starting from the root) search in the underlying suffix tree. Even in more extensive experiments, the average numbers of false positives in the CRC variants never exceed the 0.3% of the queries, suggesting that this worse-case search happens quite rarely, as we can also evince from the plots, and that 32-bits hash functions might be sufficient in most applications.

We search the same patterns for each variant; parallel to the length of the patterns, we report the average number of occurrences of each pattern as a different characterization of complexity: the running time increases as the size of the pattern grows, but when the average number of occurrences approaches one the hardness of the search problem remains the same, independently of the pattern length, and the only additional cost is the final check. Thus, after this point we do not perform tests with longer patterns: this strategy is evident in the random datasets, in which short patterns are sufficient to identify completely a single result.

The running time of our baseline (the enhanced suffix array) is mainly dependent on the alphabet size: the larger the alphabet, the longer it takes to walk down the (implicit, first-child next-sibling) suffix-tree representation. The performances of the enhanced suffix array in rand-62 and in english are similar, because both texts can be thought of as being mostly made of Latin uppercase and lowercase letters, and few other symbols – despite the alphabet in the english corpus is much larger (see Table 2), most of the other symbols rarely occur.

As we previously noted, while all the zuffix-array variants have an underlying enhanced suffix array, the size of the alphabet has little to no influence on their performance, as after the fat binary search we access a single node of the suffix tree. Therefore, zuffification can be used to improve the performance of enhanced suffix arrays on large alphabets.

The performance of the zuffix arrays is mostly dependent on the average length of the handle of the exit node. We can see indeed that all the zuffix-array variants have a similar running time across all datasets, except in the case of Fibonacci words, where the small alphabet and the enormous average handle length makes fat binary search less effective.

We can observe that zuffix arrays, at least in their fastest variants (hardware CRC and wyhash), are always faster than their non-accelerated variant (again, except in the case of Fibonacci words). Despite the enhanced suffix array being a very performant data structure, especially in small alphabet texts, zuffix arrays provide impressive speedups. In a small alphabet text like dna, the zuffix array is up to twice as fast as the enhanced suffix array. In a larger text, like dblp-xml and english, the zuffix array is up to 7 times faster than its baseline. Even when the patterns have only one occurrence in the text, the zuffix array is 1.5 to 3.5 times faster than the enhanced suffix array.

There is of course a price to pay: the zuffix array is comparable in size to the underlying enhanced suffix array in the CRC version, so it doubles the memory requirements. Nonetheless, for large alphabets we obtain a speedup of almost one order of magnitude, which we believe is a very interesting tradeoff.
9 Lessons Learned

- The z-map can make searches several times faster for medium-length patterns. As one can see from Table 3 and Figure 2–4, searches on such patterns are up to 7 times faster than on an enhanced suffix array. As the pattern length grows, however, the cost of memory access becomes dominant, so in the case of highly repetitive text with a small alphabet and long patterns the z-map theoretical improvement becomes unnoticeable.

- Hardware-supported CRCs are ideal hash functions. Software-based CRCs are too slow (indeed, by far the worst-performing hashing strategy). Ideally, a 64-bit wide hardware-supported CRC would induce fewer collisions, but even in the presence of collisions (which sometimes requires a full comparison of the pattern) hardware-supported CRCs are still the fastest option.

- Standard hash functions are sufficient for practical purposes. While CRCs are ideal in general, unless we consider pathological counterexamples, such as our Fibonacci dataset, using a standard, highly optimized hash function yields good results and sometimes provides a shorter construction time than CRCs. Construction time is theoretically quadratic, but on realistic datasets node names (and thus handles) are quite short, and modern hash functions are very fast.

- The signature approach is not useful. For the same reason (short names) the signature approach is (maybe surprisingly) slower than the memcmp approach. The problem here is that when performing the comparison, one has to hash the entire pattern, and since precomputed hash states are generated lazily, almost all the work is done at the time of the comparison, so in reality the check is closer to use time that is linear in the pattern length, rather than constant; at that point, memcmp is faster.

- Size depends on structure. The size of the z-map depends heavily on the structure of the text, in particular for large alphabets: while the number of internal nodes of a binary trie is equal to the number of leaves minus one, in the case of large alphabets it depends strictly on the structure of the text. The z-map is smaller in the case of few internal nodes with many children, which is exactly the case in which the enhanced suffix array is slower.

10 Conclusions

We have presented a detailed and carefully engineered implementation of zuffix arrays based on enhanced suffix arrays. We have shown that the z-map can improve by several times the performance of an enhanced suffix array on patterns of medium length, reaching a sevenfold speed increase in the case of large alphabets. While the space occupied by the z-map is comparable to that of the enhanced suffix array, we believe that if space is available zuffix arrays provide an interesting tradeoff between time and space, providing the practically fastest pattern lookup time.

As we remarked in the introduction, the z-map should be thought of as a speed-up gadget that can be superposed to different trie-based data structures: in the future, we plan to experiment with its application to further underlying data structures, provided that they make it possible to navigate the suffix tree.

Another interesting line of research is that of reaching different points of the space-time tradeoff by pruning the longer handles of the z-map. Indeed, the z-map works as long as it is built on a prefix of the suffix tree, but in that case, completing the search might require more than one access to the suffix tree, as it is in the case of this paper. Depending on the underlying structure, which could be, for example, compressed but very slow, choosing different prefixes would provide different tradeoffs between space and time.
References


Engineering Zuffix Arrays


Abstract

Computing a minimum path cover (MPC) of a directed acyclic graph (DAG) is a fundamental problem with a myriad of applications, including reachability. Although it is known how to solve the problem by a simple reduction to minimum flow, recent theoretical advances exploit this idea to obtain algorithms parameterized by the number of paths of an MPC, known as the width. These results obtain fast [Mäkinen et al., TALG 2019] and even linear time [Cáceres et al., SODA 2022] algorithms in the small-width regime.

In this paper, we present the first publicly available high-performance implementation of state-of-the-art MPC algorithms, including the parameterized approaches. Our experiments on random DAGs show that parameterized algorithms are orders-of-magnitude faster on dense graphs. Additionally, we present new fast pre-processing heuristics based on transitive edge sparsification. We show that our heuristics improve MPC-solvers by orders of magnitude.

1 Introduction

1.1 Motivation

A minimum path cover $\mathcal{P}$ (MPC) of a directed acyclic graph (DAG) $G = (V, E)$ is a minimum-sized set of paths covering $V$, that is, every vertex of $V$ is present in at least one path of $\mathcal{P}$. Dilworth [19] proved that the number of paths in such a set, namely the width $k$, equals the maximum number of pairwise non-reachable vertices. See Figure 1 for an illustration.

---

1 A vertex $u$ reaches a vertex $v$ if there is a path from $u$ to $v$. 

© Manuel Cáceres, Brendan Mumey, Santeri Toivonen, and Alexandru I. Tomescu; licensed under Creative Commons License CC-BY 4.0

22nd International Symposium on Experimental Algorithms (SEA 2024).

Editor: Leo Liberti; Article No. 3; pp. 3:1–3:19

Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
of these concepts. Later, Fulkerson [25] showed that the problem of finding an MPC is polynomially solvable with a reduction to maximum matching in a bipartite graph encoding the reachability relation between the vertices.

Computing an MPC has applications in many areas of computer science such as bioinformatics [39, 15, 22, 47, 10], scheduling [14, 17, 5, 49], computational logic [4, 26], distributed computing [46, 28], evolutionary computation [30], programming languages [34], databases [29], cryptography [38], and program testing [40]. Moreover, MPCs also encode the reachability between the vertices, as formally shown by the constant-time reachability index of Jagadish [29] as well as by the transitive closure algorithm of Simon [44]. As such, MPCs are fundamental objects in the problem of reachability and the applications therein.

The results of Dilworth and Fulkerson were developed in the context of partially ordered sets (posets) where the input object corresponds to a transitive DAG. The problem was later defined on general DAGs (as in this manuscript) and solved by a simple and elegant reduction to minimum flow [40], the folklore reduction. In this reduction, a minimum flow is computed on a different graph $G = (V, E)$, which is then decomposed to obtain the corresponding MPC. Hence, an MPC can be found in time $O(T_{MF}(G) + ||P||)$, where $T_{MF}$ is the time to compute a maximum flow\(^2\) and $||P||$ is the total length of the paths in the computed MPC\(^3\).

On the one hand, by using the recent breakthrough result on flows [11, 48], we can compute an MPC in almost-optimal $O(|E|^{1+o(1)} + ||P||)$-time. Although this is an impressive theoretical discovery, state-of-the-art flow algorithms rely on complex convex optimization techniques, and are far from being competitive in practice against current high-performance flow solvers (see e.g. [3] for progress in this line of research).

On the other hand, recent efforts further study the minimum flow reduction and develop algorithms parameterized by the width $k$, obtaining running times of $O(k(|V| + |E|) \log |V|)$ [23, 34, 39] and the first parameterized linear time algorithm running in time $O(k^2|V| + |E|)$ [8] and later improved to $O(k^2|V| + |E|)$ [6]. Although these approaches are beaten in the large-width regime, they have practical potential as 1) they are simple combinatorial approaches, which also facilitates their implementation, 2) the expected width of random DAGs is known to be upper-bounded by $O(\frac{\log(|V|)}{\rho})$ [2, 44], where $\rho = \frac{|E|}{|V|^2}$ is the density of the DAG, which was recently confirmed experimentally [35] and 3) the width in several applications has been observed to be even smaller [37, 9, 43].

\(^2\) In this reduction, the minimum flow problem can be reduced to maximum flow as we will explain later.

\(^3\) The MPC can be decomposed from the flow in time $O(||P|| + |E|)$ as we will explain later. Also note that $||P|| = O(k|V|)$ is a simple bound.
1.2 Contributions
We present the first open source implementations of different state-of-the-art MPC-solvers:
- The folklore reduction, which is compatible with all maximum flow and minimum cost flow solvers from the LEMON library [18] as well as our own implementations of classical maximum flow algorithms.
- The $O(k(|V| + |E|) \log |V|)$-time algorithm [23, 34, 39], which is also compatible with all the flow solvers from the previous point.
- The parameterized linear time algorithms $O(k^2|V| + |E|)$ [8], $O(k^2|V| + |E|)$ [7].

Our experiments on random DAGs show that the parameterized approaches are orders-of-magnitude faster than the folklore reduction on the fastest flow-solvers. In fact, our implementations of the parameterized approaches are able to compute MPCs on graphs with more than $10^8$ edges in less than 2 minutes. In particular, the parameterized linear time algorithms shine on dense and small-width instances and outperform all its competitors, running in less than 5 seconds. We also present new fast pre-processing heuristics based on the concept of transitive sparsification [29, 44]. By removing transitive edges, our heuristics reduce the running time of solvers by up to an order-of-magnitude.

The rest of the paper is organized as follows. Section 2 explains the algorithms in our implementations. Section 3 shows our proposed pre-processing heuristics based on transitive sparsification. Section 4 presents our experimental setup and results.

2 Flow-based MPC algorithms
All state-of-the-art MPC-algorithms are based on a simple and elegant reduction to minimum flow. Analogous to the maximum flow problem, in minimum flow [13] we are given a graph $G = (V, E)$ with a source $s \in V$ and a sink $t \in V$, and demands on the edges $d : E \to \mathbb{N}_0$. The goal is to compute an $s$-$t$-flow (or just flow) $f^* : E \to \mathbb{N}_0$ of minimum size $|f^*|$ (net flow exiting $s$), which satisfies flow conservation (the flow entering and exiting a non-source nor sink vertex is the same) and respects the demands ($f^*(e) \geq d(e)$ for all edges). For a more formal definition of these concepts we refer to [1].

2.1 The Folklore reduction
The reduction from MPC to minimum flow has been discovered and re-discovered many times in the literature (see e.g. [8]), but it can be attributed to its first public appearance in the paper of Ntafos and Hakimi [40]. Given a DAG $G = (V, E)$, we build its flow reduction as the pair $G = (V, E)$, $d : E \to \mathbb{N}_0$, where $V$ contains two copies $v^{in}$, $v^{out}$ of each vertex $v \in V$ connected by an edge with demand $d(v^{in}, v^{out}) = 1$, every other edge in $E$ has demand 0. The set $V$ also contains a global source $s$ connected to every $v^{in}$ and a global sink $t$ connected from every $v^{out}$. Finally, $E$ replicates $E$ by having an edge $(u^{out}, v^{in})$ for every edge $(u, v) \in E$. Note that $|V| = O(|V|)$, $|E| = O(|V| + |E|)$. A flow $f$ corresponds to a path cover $P$ of $G$ with $|f|$ paths. See Figure 2 for an illustration of these concepts.

Each of these paths can be obtained by decomposing one unit of flow at a time from $f$. The decomposition can be naively performed with $|f|$ graph searches in $O(|f|(|V| + |E|))$ time. We implement a faster algorithm running in time $O(||P|| + |E|)$ (see Section 2.4).

As every path cover can be interpreted as a flow, an MPC corresponds to a minimum flow $f^*$ in this network. Minimum flow on the flow reduction can be reduced to maximum flow by also providing an initial flow $f$, that is, a path cover. For every edge $e \in E$, we place it in the maximum flow instance only if $f(e) > d(e)$, in which case we define its capacity to
A feasible flow of size 4 in the flow reduction of the graph shown in Figure 1. Flow values are shown on top of the corresponding edges (0 if not present). Vertices s and t are not shown for simplicity. A decomposition of this flow produces a path cover with 4 paths, i.e. not minimum.

Figure 2: A feasible flow of size 4 in the flow reduction of the graph shown in Figure 1. Flow values are shown on top of the corresponding edges (0 if not present). Vertices s and t are not shown for simplicity. A decomposition of this flow produces a path cover with 4 paths, i.e. not minimum.

Figure 3: Residual graph of the flow in Figure 2. Direct edges are shown as solid arrows, while reverse edges as dashed arrows. A residual path is highlighted. From s and to t only the path edges are shown. Flow values on the edges are the result of using the residual path.

be \( c(e) = f(e) - d(e) \). Moreover, for every edge \((u, v) \in E\), we place its reverse edge \((v, u)\) in the maximum flow instance with capacity \( c(u, v) = |f| \). It can be shown \([39]\) that if \( f' \) is a maximum flow of this instance, then \( f^* = f' - f \) is a minimum flow of \( G, d \).

A more direct interpretation of the minimum flow problem \([8]\) defines the residual graph \( R(G, f) \), by placing every reverse edge (used to increase the flow in the opposite direction) and placing direct edges whenever \( f(e) > d(e) \) (used to decrease the flow). An \( st \)-path in \( R(G, f) \) (residual path) can then be used to decrease the flow size by one unit. See Figure 3.

In both interpretations of the problem, a minimum flow of the reduction can be obtained in time \( O(|f|(|V| + |E|)) \) by a simple Ford-Fulkerson approach \([24]\), which finds \( O(|f|) \) residual paths. Since there is always a path cover that uses \( |V| \) paths to cover every vertex (one path per vertex, the naive solution), the previous approach runs in \( O(|V||V| + |E|) \) time.

### 2.2 Greedy solution

Felsner et al. \([23]\) proposed a greedy heuristic to compute a chain decomposition of a poset. They iteratively extract the longest chain of elements from the remaining poset. They proved, with analogous arguments to those of the greedy set cover logarithmic approximation \([12]\), that the number of chains extracted is bounded by \( O(k \log |V|) \). Later, Kowaluk et al. \([34]\) showed that the same principle can be applied to general DAGs by finding the path covering the most uncovered vertices. They showed that these paths can be found in a DAG by a reduction to shortest path, which was later simplified by Mäkinen et al. \([39]\) with a dynamic programming solution. As such, computing the greedy solution and using it in the flow reduction derives a \( O(k(|V| + |E|) \log |V|) \)-time algorithm for MPC.

---

\[4\] Flow in reverse edges is interpreted as negative flow in the opposite direction.
2.3 Parameterized linear time algorithms

Recently, Cáceres et al. [8] proposed a slightly different method to compute an MPC using the flow reduction. In their method, the vertices are processed one by one in topological order [31, 45] and an MPC of the (graph induced by the) already processed vertices is computed at each step. More specifically, if \( \mathcal{P} \) is an MPC of the first vertices in topological order and \( v \) is the next vertex to process, the solution \( \mathcal{T} = \mathcal{P} \cup \{(v)\} \) is used as initial solution of the flow reduction. A nice property of this incremental framework is that the size of an MPC of the current iteration is either \(|\mathcal{T}| = |\mathcal{P}| + 1\) or \(|\mathcal{P}|\), and thus only one traversal of the residual suffices to compute it. Note that this a simplification of the framework. In practice, the algorithms maintain a flow \( f^* \) on the flow reduction of the first vertices (representing the MPC \( \mathcal{P} \)). When the next vertex \( v \) is processed, the flow reduction is updated by adding vertices \( v^{in} \) and \( v^{out} \), and their corresponding edges (see Section 2.1). The flow \( f^* \) is also updated to \( f^*(s, v^{in}) = f^*(v^{in}, v^{out}) = f^*(v^{out}, t) = 1 \) (representing \( \mathcal{T} \)), which is then used to look for a residual path, possibly modifying \( f^* \) again.

By considering simple graph traversals of the residual, this approach runs in time \( O(|V|(|V| + |E|)) \). However, Cáceres et al. [8] combine transitive sparsification of edges with a special layered traversal of the residual to obtain a linear dependency in the number of edges and a factor \( k^3 \) dependency in the number of vertices for a total running time of \( O(k^3|V| + |E|) \).

Layered traversal. The algorithm assigns a level \( \ell : V \rightarrow \{0, \ldots, |\mathcal{P}|\} \) to every vertex in the flow reduction. The level assignment maintains the property that paths in the residual graph are sequences of vertices with non-increasing levels, which allows to perform the traversal for the search of a residual path in a layered\(^6\) manner. The residual graph is traversed from the highest reachable layer until the lowest reachable layer (or until a residual path is found). To perform this layered traversal, the algorithm uses \(|\mathcal{P}| + 1\) FIFO queues (one per layer), each of which performs a BFS from the highest-reachable layer down to the lowest-reachable layer. Algorithm 4 in the Appendix shows the corresponding pseudocode of the traversal.

After the layered traversal finishes, the algorithm updates the flow and level assignment to maintain the algorithm’s invariants [8]. The flow is only updated if the traversal finds a residual path, in such a case, the flow in direct edges of the path is decreased by one and the flow in (the reverse of) reverse edges is increased by one, which decreases the total flow by one. As for the level assignment, if the lowest visited level is \( l \), then all visited vertices change their level to \( l \) while the level of \( v^{out} \) is set to \( l + 1 \). Additionally, if there is no flow from layer \( l \) exiting directly to \( t \), then the algorithm performs a merge of layer \( l \). For an explanation of the merge procedure, we refer to the original publication [8] and our code.

Transitive sparsification. An edge \((u, v)\) is transitive if there exists another path from \( u \) to \( v \). Transitive edges can be safely removed from the DAG when computing an MPC, as removing these edges preserves the reachability relation between the vertices and hence, by Dilworth’s theorem, the width. A transitive sparsification is both a spanning subgraph with the same reachability relation as \( G \) as well as the process to obtain such a subgraph. The \( O(k^3|V| + |E|) \)-time algorithm sparsifies the number of incoming edges to each vertex to \( O(k) \). The authors use a simple idea first proposed by Jagadish [29] and Simon [44] on

\(^5\) As topological sorting algorithms run in linear time, we assume that such order is given as input. In our implementation we use the faster DFS-based algorithm of Tarjan [45].

\(^6\) A layer is a maximal set of vertices with the same level.
posets\(^7\): if several incoming edges to \(v\) come from the same path \(P\), then all these edges, except maybe the last, are transitive. Conversely, if several outgoing edges from \(u\) go to the same path \(P\), then all these edges, except maybe the first, are transitive. See Figure 4. The algorithm of Cáceres et al. [8] uses this idea and the MPC \(P\) from the previous iteration to sparsify the edges to the current vertex \(v\) to at most \(|P| \leq k\). To perform this sparsification efficiently, the algorithm requires that every vertex stores the id of one path that contains such vertex, which is achieved by maintaining a path decomposition of the minimum flow \(f^*\), that is \(P\).

The \(O(k^2|V| + |E|)\)-time algorithm. The same authors later improved the running time of their algorithm [7] by shaving a \(k\)-factor from the dependency on the number of vertices. They noted that it is not necessary to maintain the ids of all paths containing a vertex during the algorithm, but that it suffices to maintain only one of those ids, and the MPC can be retrieved at the end by performing only one decomposition. To achieve this, they identified a set of antichain vertices separating consecutive layers: as these vertices form an antichain, each of those must be covered by a different path, and in fact the algorithm covers each of these vertices with exactly one path (we refer to the original publication [7] for details). As such, it suffices that every vertex points back to (one of) the corresponding antichain vertex on its layer, these pointers are called back links (bl in the pseudocode for short). As opposed to an entire decomposition, back links can be maintained by only decomposing the vertices in layer \(l\) (lowest visited level), and then fixing the back links of vertices in lower layers (in constant time per vertex of level \(>l\)). Algorithm 1 implements the ideas in [8, 6].

2.4 The decomposition algorithm

As mentioned earlier, the last step of all MPC-solvers, as well as intermediate steps of the parameterized linear time algorithms, require to decompose the flow \(f^*\) into an MPC. A naive solution extracts one path at a time in total \(O(|f^*|(|V| + |E|))\) running time. We instead implement an algorithm that runs in time \(O(||P|| + |E|)\)\(^8\). Such an improvement was first described by Kogan and Parter [32]. Here we use the version of Cáceres [6]. The algorithm first removes all 0-flow edges in time \(O(|E|)\) and then processes the vertices in topological order. When processing vertex \(v\), it iterates through each in-neighbor \(u\) and places \(v\) after \(u\) in \(f^*(u,v)\) different paths\(^9\). The total running time is \(\sum_{v\in V, (u,v)\in E} f^*(u,v) = O(||P||)\).

---

\(^7\) While Jagadish presented the idea on transitive DAGs, Simon worked on general DAGs but using a path decomposition/partition instead of a path cover.

\(^8\) Note that \(||P|| = O(|f^*| \cdot |V|)\), and thus this approach removes a factor \(|f^*|\) from \(|E|\).

\(^9\) This can be done in constant time per path by iterating through the list of paths of \(u\).
Algorithm 1 One iteration of the $O(k^2|V| + |E|)$-time algorithm. The pseudocode shows the steps of the algorithm when processing vertex $v$. Function $id$ returns the path id stored in an antichain vertex. Function $bl$ returns the corresponding antichain vertex (back link). Notation $u_Pw$ indicates the decomposed path connecting antichain vertices $u$ and $w$. For more details we refer to the original publication [7] and our code.

```plaintext
// Add $v$ to flow reduction
$V ← V \cup \{(v^{in}, v^{out})\}$
$E ← E \cup \{(s, v^{in}), (v^{in}, v^{out}), (v^{out}, t)\}$
// Create initial solution $T$
$f^*(s, v^{in}), f^*(v^{in}, v^{out}), f^*(v^{out}, t) ← 1$
// Sparsify edges incoming to $v$
$T ← \{ (u, v) \mid \text{sur}[id(bl(u))] = \max(\text{sur}[id(bl(u))], u) \}$
$\text{sur}[id(bl(u))] ← \text{max}(\text{sur}[id(bl(u))], u)$
$E ← E \cup \{|u,v|\}$
for Decomposed path $u_Pw$ do
for Decomposed path $u_Pw$ do
// Fix backlinks in higher layers
for $u, w \in V, \ell(w^{out}) > \ell(v^{out}) > \ell(w^{in})$ in top. order do
if $D ≠ \emptyset$ then
for $(u, w) \in E$ do
if $(u, w) \in E$ then
else $f^*(u, w) ← f^*(u, w) - 1$
else $f^*(u, w) ← f^*(u, w) + 1$
if $\text{nl}(x) = w$ do
if $\text{nl}(x) = w$ do
$\ell(x) ← w$
$\ell(x) ← w$
```

3 Fast pre-processing sparsification heuristics

In this section, we present two transitive sparsification heuristics. Recall that a transitive sparsification removes transitive edges, making the input graph sparser. These heuristics are intended to be used as pre-processing steps of MPC-solvers to speed up their computation. As such, it is very important that they run fast compared to the MPC-solver. In fact, we ensure that their worst-case running time is upper-bounded by the running time of state-of-the-art solvers. Both of our heuristics use paths to sparsify the incoming/outgoing edges to/from a vertex as done by the parameterized linear time algorithms.

DFS sparsification. Our first sparsification heuristic uses the root-to-leaf paths of a DFS-spanning tree. A first naive implementation of this idea processes each of these paths to sparsify the incoming edges. However, this approach runs in time proportional to the total length of the root-to-leaf paths, which can be $\Omega(|V|^2)$. Instead, our simple solution runs in time $O(|V| + |E|)$ as it is implemented on top of a normal recursive DFS traversal. Algorithm 2 shows the corresponding pseudocode.

The main idea behind this algorithm is to use the DFS recursion itself as DFS paths. For this, it stores the preorder of each vertex visited (in $dfs$$_{pre}$) as well as the maximum preorder value observed of an in-neighbor (in $last$_$reach$). When processing an edge $(v, w)$
**Algorithm 2** DFS sparsification heuristic.

1. $E' \leftarrow \emptyset$
2. $S \leftarrow \emptyset$
3. $\text{dfs}_\text{pre}[v] \leftarrow 0$ for $v \in V$
4. $\text{next}_\text{pre} \leftarrow 1$
5. $\text{last}_\text{reach}[v] \leftarrow 0$ for $v \in V$
6. for $v \in V$ in top. order do
7.   if $v \not\in S$ then $\text{dfsSp}(v)$
8. return $G' = (V, E')$

Function $\text{dfsSp}(v)$:

9. $S \leftarrow S \cup \{v\}$
10. $\text{dfs}_\text{pre}[v] \leftarrow \text{next}_\text{pre}$
11. $\text{next}_\text{pre} \leftarrow \text{next}_\text{pre} + 1$
12. for $w \in \text{N}^+(v)$ in top. order do
13.   if $w \not\in S$ then $\text{dfsSp}(w)$
14.   if $\text{last}_\text{reach}[w] < \text{dfs}_\text{pre}[v]$ then
15.     $E' \leftarrow E' \cup \{(v, w)\}$
16.     $\text{last}_\text{reach}[w] \leftarrow \text{dfs}_\text{pre}[v]$

**Algorithm 3** Greedy sparsification heuristic.

1. $E' \leftarrow E$
2. $P \leftarrow \emptyset$
3. while $P$ is not a path cover do
4.   $P^* \leftarrow$ path of $(V, E')$ with most uncovered
5.   $P \leftarrow P \cup \{P^*\}$
6.   $R \leftarrow \emptyset$
7. for $v \in P^*$ do
8.   for $u \in \text{N}^-(v)$ in $(V, E')$ do
9.     if $u \in R$ and $(u, v) \not\in P$ then
10.    $E' \leftarrow E' \setminus \{(u, v)\}$
11.    $R \leftarrow R \cup \{u\}$
12. return $G' = (V, E'), P$

(assuming traversing $w$), if the observed preorder value of an in-neighbor of $w$ is bigger than the preorder of $v$ ($\text{last}_\text{reach}[w] > \text{dfs}_\text{pre}[v]$), then the edge is transitive and it is not added to the sparsification, as there is a vertex further down the DFS-tree also with an edge to $w$ (the one with preorder value $\text{last}_\text{reach}[w]$). Conversely, among all vertices in a DFS root-to-leaf path with an edge to $w$, the only edge that is not sparsified is the one with the largest preorder value, that is, the one closest to the leaf.

**Greedy sparsification.** Our second sparsification heuristic also outputs the greedy initial solution explained in Section 2.2. It uses the $O(k \log |V|)$ paths from the greedy solution to sparsify outgoing edges. As such, this heuristic sparsifies the edges to $|E'| = O(k|V| \log |V|)$. Since this algorithm computes the greedy solution, its worst-case running time is also $O(k(|V| + |E|))$. However, we implemented a non-trivial practical improvement where each extracted path is immediately used to sparsify, and thus the following paths are extracted from a sparser graph. Algorithm 3 shows the corresponding pseudocode. The main novelty of this approach is that greedy paths are computed at the same time that the DAG is being sparsified, resulting in a faster initial solution computation. Note that the algorithm does not sparsify an edge if this is present in the greedy path cover, however, there are at most $|P| = O(k|V| \log |V|)$ such edges.

4 Experiments and Results

4.1 Implementations

Our code was written in C++ and it can be found at https://github.com/algbio/PerformanceMPC under the GNU General Public License v3.0. The code is compatible with all maximum flow and minimum cost flow solvers from the LEMON library [18], which
are known to be the fastest publicly available flow solvers. For an input, we provide the best times among the solvers in the library as lemon. For a cleaner and fairer comparison, we re-implemented the following well-known maximum flow solvers:

- **BFS**: Edmonds-Karp algorithm [21], which finds residual paths using breath-first search.
- **Blocking**: Dinitz’ algorithm [20], which uses blocking flows.

All solvers can start from one of the following initial solutions (path covers):

- naive: |V| paths, each covering a different vertex.
- greedy: \(O(k \log |V|)\) paths based on greedy set cover [23, 34, 39].

After running the flow solver all our implementations run the same \(O(||P|| + |E|)\)-time decomposition routine to obtain the corresponding MPC \(P\) (see Section 2.4).

Our code also implements the parameterized linear time algorithms:

- \(k_3\): The first parameterized linear time algorithm running in time \(O(k^3|V| + |E|)\) [8].
- \(k_2\): A later improvement over \(k_3\) running in time \(O(k^2|V| + |E|)\) [7].

All these implementations constitute the state-of-the-art algorithms for MPC. To the best of our knowledge there are no other publicly available fast MPC-solver’s implementations. Most publicly available MPC-solvers use the slower reduction to bipartite maximum matching, and thus also need to compute the transitive closure\(^{10}\). Mäkinen et al. [39] were the first to implement the greedy-based approach, which was later improved by Ma et al. [37] using Dinitz’ algorithm for finding residual paths: these implementations correspond to our DFS greedy and Blocking greedy, respectively. Finally, for all our MPC-solvers we also implemented our two sparsification heuristics from Section 3: dfs-sp and greedy-sp.

### 4.2 Setup

The experiments ran on an isolated Intel(R) Xeon(R) CPU E5-2670 @ 2.6 GHz with 64GB of RAM, running Almalinux 8.4 (64bit, kernel 4.18.0). The code was compiled using gcc version 8.5.0 with optimization flag -O3. We measure user time using the sys/resource.h Unix library. We report the average value of 10 repetitions of each experiment. We used a timeout of 10 minutes for each experiment.

### 4.3 Datasets

**Random DAG.** For a fixed value of \(N\) and \(M\), we generate a random DAG with \(N\) vertices and \(M\) edges. The generation procedure first fixes a topological order of the \(N\) vertices. Then, it generates \(M\) different pairs of vertices and interprets them as edges directed according to the topological order. We vary \(N \in 10,000 \times \{1, 2, \ldots, 5\}\). For space constraints reasons, we show the results for \(N = 50,000\) and vary \(M \in \{32, 768 = 2^{15}, 2^{16}, \ldots, 2^{27} = 134,217,128\}\) to observe the behavior at different densities (other values of \(N\) can be found in the Appendix). This dataset corresponds to the random DAG model proposed by Barak and Erdös [2]. We use this dataset to compare general performance. Table 1 in the Appendix shows the width of DAGs in this dataset. Note that the width decreases with the number of edges of the Random DAG. Indeed, the expected width of a Random DAG of parameters \(N\) and \(M\), is upper bounded by \(O\left(\frac{\log (pN)}{p} \right)\) [44], where \(p = \frac{M}{\binom{N}{2}}\) is the density of the DAG. Moreover, Kritikakis and Tollis [35] recently showed that experimentally the width is proportional to \(1/p\).

\(^{10}\)The densest spanning supergraph having the original graph as a transitive sparsification.
Path Partition. For a fixed value of $N$, $M$ and $K$, we generate the previously described Random DAG with $N$ vertices and $M$ edges. Then, we divide the $N$ vertices into $K$ parts by placing each vertex on a uniformly random chosen part. Finally, we add the corresponding $N - K$ edges (in topological order) so that each part is a path in the DAG. As such, the graph’s width is at most $K$. We fix $N = 50,000$ and vary $M \in \{32, 768 = 2^{15}, \ldots, 2^{27} = 134, 217, 128\}$ as before and , $K = \{21 = \lfloor 2^1 \cdot \ln N \rfloor, \ldots, \lfloor 2^4 \cdot \ln N \rfloor = 173\}$, we only show the results for $K = 173$ due to space constraints (other values of $K$ can be found in the Appendix). We use this dataset to study the performance on small-width instances. Table 2 in the Appendix shows the number of edges and width of DAGs in this dataset. We note that our Path Partition dataset is equivalent to the “Path-Based Model” of Lionakis et al. [36] and a generalization of the “Random arcs $k$-path Model” of Paavilainen [42].

Transitive Closure. For a fixed value of $N$ and $M$, we generate a Random DAG with $N$ vertices and $M$ edges. Then, we compute its transitive closure. We fix $N = 10,000$ and vary $M \in \{8, 192 = 2^1, \ldots, 2^{27} = 8, 388, 608\}$. We use this dataset to study the performance on posets and the behavior of transitive sparsification heuristics. Table 2 in the Appendix shows the number of edges and width of DAGs in this dataset. Note that the width distribution of Random DAG is not affected as adding transitive edges does not change the width.

4.4 Results

Figure 5 shows the running time of the MPC-solvers on the Random DAG dataset (for other values of $N$ see Figure 9 in the Appendix). Solvers starting from a naive solution are depicted with a solid line joining the corresponding data points. The maximum flow-based solvers show a polynomial dependency in the number of edges of the input graph, with Blocking naive being the fastest (as predicted by theory as it uses a faster flow algorithm) followed by DFS naive and then by BFS naive. These results suggest that 1) the more complex Blocking algorithm pays off, as each step significantly reduces the path cover size, and that 2) although BFS ensures a polynomial running time for maximum flow, in the case

---

11We consider k3 and k2 “to start from a naive solution” since, at each step, these consider the next vertex as a single path.
of MPC this is unnecessary as $k \leq |V|$ and DFS performs better in practice as residual paths are quickly found. Moreover, these solvers run out of time (> 10 mins.) after $|E| = 2^{24}$, $2^{20}$ and $2^{18}$, respectively. Solvers starting from a greedy solution are shown with a dotted line. The solvers show a much faster running time, which stands below the 2 mins. irrespective of the number of edges. As such, on dense graphs, these approaches are orders-of-magnitude faster that their naive counterparts. In this case, the difference between the different solvers is subtle, as substantially less residual paths must be found to transform the $O(\log |V|)$-approximation to an MPC, and indeed DFS beats the machinery of Blocking from $|E| \geq 2^{22}$ and $k \leq 494$. The running time of $k3$ decreases with the number of edges. This is explained by the linear dependency on the number of edges: these algorithms process the edges, in constant-time, only during the initial edge sparsification, whereas vertices are charged with all the remaining machinery of the approach. This amounts to $O(k^3)$ and $O(k^2)$ per vertex, respectively, which is known to decrease with increasing density [44, 35]. Both solvers run on less than 2 mins. on every graph, and outperform the maximum flow-based solvers on dense graphs, from $|E| \geq 2^{20}$ in the case of $k2$ and from $|E| \geq 2^{23}$ in the case of $k3$, being almost
two orders of magnitude faster on the densest instance (|E| = 2^{27}). Finally, it is worth mentioning that k2 outperforms k3 on every instance tested, which shows that the more complex routines of the k2 algorithm, as implemented in this work, manage to effectively shave a factor $k$ from the running time. In practical terms, the time saved by avoiding the full decomposition is larger that the time required to perform these savings. Finally, the best of lemon dominates the naive versions of our solvers but it is dominated in the dense regime, by both the greedy versions and the parameterized linear time algorithms. In the sparse regime lemon corresponds to the implementations Preflow [27] and CapacityScaling [21], whereas in the dense regime is dominated by CostScaling [41] as well as NetworkSimplex [16].

Figure 6 shows the running time of the solvers on the Path Partition dataset (for other values of $K$ see Figure 10 in the Appendix). The behavior is very similar to the Random DAG dataset. In this case, DFS beats Blocking in sparse and very sparse graphs in greedy and naive, respectively. For all graphs in the dataset, the solvers $k_3$ and $k_2$ run in no more than 15 secs. and 5 secs., respectively.

Pre-processing. We exclude $k_3$ and $k_2$ from this comparison as using a pre-processing edge sparsification is counterproductive (perform internal sparsification). For space constraints we only show the effect of pre-processing on one MPC-solver.

Figure 7 shows the running time of Blocking greedy on the Transitive closure dataset and different combinations of pre-processings dfs-sp and greedy-sp (for other MPC-solvers see Figure 8 in the Appendix). We note that for $M \geq 2^{19}$ the number of edges in the corresponding graphs is larger than in the densest instance of the previous datasets, as such we call these graphs dense. On dense graphs, dfs-sp roughly decreases the running time in half, while greedy-sp reduces the running time by one order-of-magnitude. When using both heuristics greedy-sp and dfs-sp, we perceive a combined positive effect until $M \leq 2^{19}$. For denser graphs, performing both sparsifications does not pay off as greedy-sp is able to sparsify more edges (recall that greedy-sp sparsifies the edges to $|E'| = O(k|V|\log |V|)$), but it does not affect the running time significantly either. On non-dense graphs ($M < 2^{19}$), applying both sparsifications dominates and it is up to 4 times faster than plain greedy.

Even though our sparsification heuristics show a clear improvement on dense graphs, these improved versions are still outperformed by the parameterized approaches.

5 Conclusions and Future Work

We presented the first high-performance implementation of state-of-the-art MPC algorithms and showed that approaches parameterized by the width dominate the practical performance landscape on different kinds of random graphs. In particular, the parameterized linear time algorithms [8, 7] shine on small-width instances, being orders-of-magnitude faster. Recent works [33, 6] circumvent the $\Omega(k|V|)$ lower bound by computing a minimum chain cover instead. In practice, it is interesting to test if these algorithms are effectively faster than our MPC-solvers or if these ideas can be used to improve the performance of our implementations.

We also presented two new pre-processing fast heuristics based on transitive sparsification and showed how they improve the running time by an order-of-magnitude.

An important application of MPC is reachability. In particular, it is known (see e.g. [35]) how to compute a constant-time reachability index of size $O(k|V|)$ in time $O(k|E'|)$, where $|E'|$ is the number of edges in the sparsest transitive sparsification, also known as transitive

---

12 There are instances with $||P|| = \Omega(k|V|)$ [6].
reduction. This result directly derives parameterized linear time solutions for the problems of constant-time reachability, transitive closure and transitive reduction, which can be implemented and compared against state-of-the-art solutions for those problems.

Finally, one algorithm we have not implemented is the $O(k^2 |V| \log |V| + |E|)$-time approach of Cáceres et al. [8, Theorem 1.1] as this was later outperformed by the $O(k^2 |V| + |E|)$-time algorithm [7]. However, this divide-and-conquer approach is simple to parallelize [8, Theorem 1.2] and thus it could outperform our implementations when run on multiple processors.

References


A Extra Algorithms and Figures

Algorithm 4 Layered traversal for the search of a residual path in $R(G, f)$. The algorithm returns a residual path (if one is found) as well as the set of visited vertices during the traversal.

```plaintext
Function layeredTraversal($G, f, \ell, v$):

  // Visited vertices
  \( S \leftarrow \{v^{\text{in}}\} \)
  // For each \( j \in \{0, \ldots, |f|\} \)
  \( Q_j \leftarrow \{u^{\text{out}} \mid (u, v) \in E \land \ell(u^{\text{out}}) = j\} \)
  for \( j \leftarrow |f| \) down to 0 do
    while \( Q_j \neq \emptyset \) do
      Remove \( u \) from the front of \( Q_j \)
      \( S \leftarrow S \cup \{u\} \)
      for \( v \in N^+(u) \) in \( R(G, f) \) do
        if \( v = t \) then
          return Residual path \( D, S \)
        if \( v \notin S \) then
          Add \( v \) to the back of \( Q_{\ell(v)} \)
      end
    end
  end
  return \( \emptyset, S \)
```

B Additional Experimental Data and Results

Table 1 Width $k$ for dataset Random DAG ($N = 50,000$) and different values of parameter $M$.

| $M = |E|$ | $2^{15}$ | $2^{16}$ | $2^{17}$ | $2^{18}$ | $2^{19}$ | $2^{20}$ | $2^{21}$ | $2^{22}$ | $2^{23}$ | $2^{24}$ | $2^{25}$ | $2^{26}$ | $2^{27}$ |
|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
| Width $k$ | 31,282  | 22,586  | 13,913  | 7,418   | 3,768   | 1,922   | 980     | 494     | 260     | 134     | 75      | 39      | 22      |
**Table 2** Number of edges |E| and width k for datasets Path Partition and Transitive Closure and different values of parameter M.

<table>
<thead>
<tr>
<th>M</th>
<th>Path Partition N = 50,000, K = 173</th>
<th>Transitive Closure N = 10,000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>E</td>
</tr>
<tr>
<td>2^{13}</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2^{14}</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>2^{15}</td>
<td>82,595</td>
<td>173</td>
</tr>
<tr>
<td>2^{16}</td>
<td>115,363</td>
<td>173</td>
</tr>
<tr>
<td>2^{17}</td>
<td>180,899</td>
<td>173</td>
</tr>
<tr>
<td>2^{18}</td>
<td>311,971</td>
<td>173</td>
</tr>
<tr>
<td>2^{19}</td>
<td>574,115</td>
<td>173</td>
</tr>
<tr>
<td>2^{20}</td>
<td>1,098,403</td>
<td>171</td>
</tr>
<tr>
<td>2^{21}</td>
<td>2,146,979</td>
<td>165</td>
</tr>
<tr>
<td>2^{22}</td>
<td>4,244,131</td>
<td>140</td>
</tr>
<tr>
<td>2^{23}</td>
<td>8,438,435</td>
<td>111</td>
</tr>
<tr>
<td>2^{24}</td>
<td>16,827,043</td>
<td>79</td>
</tr>
<tr>
<td>2^{25}</td>
<td>33,604,259</td>
<td>54</td>
</tr>
<tr>
<td>2^{26}</td>
<td>67,158,691</td>
<td>34</td>
</tr>
<tr>
<td>2^{27}</td>
<td>134,267,555</td>
<td>21</td>
</tr>
</tbody>
</table>

**Figure 8** Running time of other solvers in dataset Transitive Closure with different combinations of pre-processings dfs-sp and greedy-sp.
Figure 9 Running time of MPC-solvers in dataset Random DAG, for different $N$. 
Figure 10 Running time of MPC-solvers in dataset Path Partition, for different $K$. 
Separator Based Data Reduction for the Maximum Cut Problem

Jonas Charfreitag
Institute of Computer Science, University of Bonn, Germany

Christine Dahn
Institute of Computer Science, University of Bonn, Germany

Michael Kaibel
Institute of Computer Science, University of Bonn, Germany

Philip Mayer
Institute of Computer Science, University of Bonn, Germany

Petra Mutzel
Institute of Computer Science, University of Bonn, Germany

Lukas Schürmann
Institute of Computer Science, University of Bonn, Germany

Abstract
Preprocessing is an important ingredient for solving the maximum cut problem to optimality on real-world graphs. In our work, we derive a new framework for data reduction rules based on vertex separators. Vertex separators are sets of vertices, whose removal increases the number of connected components of a graph. Certain small separators can be found in linear time, allowing for an efficient combination of our framework with existing data reduction rules. Additionally, we complement known data reduction rules for triangles with a new one.

In our computational experiments on established benchmark instances, we clearly show the effectiveness and efficiency of our proposed data reduction techniques. The resulting graphs are significantly smaller than in earlier studies and sometimes no vertex is left, so preprocessing has fully solved the instance to optimality. The introduced techniques are also shown to offer significant speedup potential for an exact state-of-the-art solver and to help a state-of-the-art heuristic to produce solutions of higher quality.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization; Mathematics of computing → Solvers; Theory of computation → Network optimization

Keywords and phrases Data Reduction, Maximum Cut, Vertex Separators

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.4

Supplementary Material
Software (Source Code): https://github.com/CharJon/MaxCutDataReduction
archived at swh:1:dir:2688a073fd6e4ad2414d9db52d40757da911a7bd7

Funding This work was partially funded by the German Research Foundation (DFG) EXC-2047/1 - 390685813 and FOR-5361 - 459420781, and by the BMBF (Germany) and state of NRW as part of the Lamarr-Institute, LAMARR22B.

Acknowledgements We thank Konstantinos Papadopoulos for his contribution to this work.

1 Introduction
The maximum cut problem (MaxCut) asks for a vertex-bipartition of a given edge-weighted input graph $G$ maximizing the value of the cut. The cut is defined as the set of edges connecting vertices from different partitions and its value is the sum over all weights of edges
forming the cut. MaxCut is a fundamental problem in computer science and is one of the NP-hard problems on Karp’s famous list [28] of 21 problems. The research on MaxCut is strongly motivated by its applications in, e.g., image processing [37] or VLSI design [5], and its importance for quantum annealing [25]. To tackle general graphs encountered in these applications, exact state-of-the-art solvers like McSparse [8] and QuBowl [35] employ a multitude of different techniques. One of those is preprocessing (also called presolving).

Preprocessing algorithms make use of decomposition techniques and data reduction rules without sacrificing optimal solutions. The former split the given input into several independent smaller instances, and the latter reduce the input size. Efficient and effective data reduction rules for MaxCut have been designed in e.g. [29], [14] and [35]. They have been proven to play a crucial role in solving difficult instances for MaxCut.

MaxCut, as a combinatorial optimization problem on graphs, allows for a natural decomposition of the input into its biconnected components [21]. More sophisticated techniques based on divide-and-conquer algorithms using vertex separators have been used for special cases such as graphs with bounded treewidth [43] or graphs not contractible to $K_5$ [4] or to $K_{3,3}$ [10]. We call a set of vertices a vertex separator of a graph if its removal from the graph increases the number of connected components.

Our Contribution

We derive new exact data reduction techniques for MaxCut and evaluate their performance in a sophisticated experimental study. In detail:

1. We design a framework for exact data reduction, making use of vertex separators, which have not been made practical use of in the context of data reduction for MaxCut before. Given a vertex separator, we provide a general characterization of when and how it can be exploited for data reduction.

2. For small separators, which can be found efficiently, we derive concrete and effective rules through our framework. For certain structures in the input, our framework also allows to generalize existing rules.

3. For triangles in the input graph we introduce a new data reduction rule, complementing two existing ones.

4. We carefully engineer an algorithm consisting of state-of-the-art techniques and our new ones. We evaluate it on well-established benchmark graphs and compare the results with the state-of-the-art. Additionally, we investigate the influence of our algorithm’s individual components and their influence on the solution quality of a state-of-the-art heuristic and runtime of a state-of-the-art exact solver.

Outline

The remainder of the paper is structured as follows: In Section 2 we introduce some basic notation relevant for the theory of our paper, along with an overview of the related literature in the area of data reduction and MaxCut. Section 3 first describes existing data reduction rules that exploit weights of edges in detail. Afterwards, our new one for triangles is presented. Section 4 focuses on vertex separators and generalizes existing data reduction rules through the lens of the new framework and presents our new rules derived from this framework. The algorithm we engineered is described in Section 5 and evaluated extensively in Section 6. Section 7 concludes our findings and results.
2 Preliminaries

2.1 Notation

Throughout the paper, we consider simple undirected graphs \( G = (V, E, w) \) with edge weights \( w : E \to \mathbb{R} \). For better readability we write \( w(e) = w_e \) and \( w(\{u, v\}) = w_{uv} \). \( N(v) \) for \( v \in V \) denotes the neighborhood of \( v \) in \( G \) and \( d(v) \) its degree, so \( d(v) = |N(v)| \). For sets \( S \subset V \) we write \( N(S) \) to capture the union of all neighborhoods of vertices in \( S \) excluding all vertices in \( S \). For a set of vertices \( V' \subset V \) we denote by \( G[V'] \) the subgraph induced by \( V' \). We define the contraction of an edge \( e = \{u, v\} \) as replacing \( u \) and \( v \) by a new vertex \( a \). For every vertex \( b \in N(\{u, v\}) \) an edge \( \{a, b\} \) is added with weight \( w_{ab} + w_{vb} \), if both \( u \) and \( v \) are adjacent to \( b \) and the same weight as the edge between \( u \) or \( v \) and \( b \) otherwise.

A connected component of a graph \( G \) is a maximal subgraph of \( G \) in which any pair of vertices can reach each other by a walk. If a graph has exactly one connected component, it is called a connected graph. A graph with more than \( k \geq 1 \) vertices is called \( k \)-connected if it is connected, and the deletion of an arbitrary set of up to \( k - 1 \) vertices does not change this property. For a graph \( G \) a (non-empty) set of vertices \( S \subset V \), whose removal increases the number of connected components by at least one, is called a \( k \)-separator with \( k = |S| \).

A bipartition \( P \) of the vertices of a graph \( G = (V, E) \) is a pair \( P = (S_1, S_2) \), with \( S_1 \cap S_2 = \emptyset \) and \( S_1 \cup S_2 = V \). We also define a partial bipartition \( P' = (S'_1, S'_2) \) of the vertices, with again \( S'_1 \cap S'_2 = \emptyset \), but \( S'_1 \cup S'_2 \subset V \). Every vertex bipartition \( P = (S_1, S_2) \) induces a set of edges, called a cut \( \delta_G(P) = \{ \{u, v\} \in E \mid u \in S_1 \land v \in S_2 \} \). If the underlying graph is clear from the context, we write \( \delta(P) \). The value of a cut \( \delta(P) \) in graph \( G = (V, E, w) \) is given by \( \sum_{e \in \delta(P)} w_e \). We write \( \Delta(G) \) for the value of a maximum cut in \( G \). For a given partial bipartition \( P' = (S'_1, S'_2) \) we write \( \Delta(G, P') \) to denote the maximum value of all cuts in \( G \) respecting \( P' \). A cut respects a partial bipartition if none of its edges connects vertices from the same partition in \( P' \).

Data Reduction. A data reduction rule is applied to a weighted graph \( G = (V, E, w) \) in order to reduce its size or complexity. Independent of the concrete technique, a data reduction rule is said to be valid if the new instance \( G' = (V', E', w') \) retains an optimal solution of the original instance. We call a valid data reduction feasible if it can be applied in polynomial time. To capture the validity of graph transformations more formally, we introduce the following condition:

**Condition 2.1 (Valid Data Transformation).** Let \( G = (V, E, w) \) be a simple undirected weighted graph. Any MaxCut-valid data transformation transforms an input graph \( G \) into \( G' \) and outputs some constant offset \( \beta \in \mathbb{R} \). The MaxCut-value in \( G \) has to be the same as in \( G' \), apart from the offset \( \beta \): \( \Delta(G) = \Delta(G') + \beta \).

A (valid) data transformation is called a (valid) data reduction, if it reduces the number of vertices or edges of the input graph. Note: We clearly distinguish between kernelization and data reduction. Kernelization in the standard literature is only defined in the context of a problem specific input parameter (like treewidth, see e.g. [15]) and a kernelization algorithm is required to produce a so-called kernel, whose size is bounded by a polynomial function in the parameter. As the algorithms considered here do not rely on any input-dependent parameter, we will solely use the term data reduction.
2.2 State-of-the-Art

The literature on the three main topics of this paper, namely MaxCut, data reduction and separators, is extensive. Therefore we will only present the most relevant related work in this section, starting with MaxCut.

For general graphs MaxCut is NP-hard [28] and APX-hard [34], even if all edge weights are restricted to 1. For graphs with only non-negative edge weights Goemans and Williamson [16] suggest an algorithm with an approximation guarantee of 0.87856. Polynomial time algorithms for MaxCut have been designed for certain classes of input graphs, e.g., planar graphs [20, 4, 30] and graphs not contractible to $K_5$ [4] or $K_{3,3}$ [10]. Other tractable classes of graphs require specific edge weight distributions (see, e.g., [32]). For some of those divide-and-conquer paradigms have been used. E.g., for a given planar graph, Barahona [4] suggested transforming the maximum cut problem into a Chinese postman problem in the dual, which was then solved using the planar separator theorem. For the more general class of graphs not contractible to $K_5$, Barahona [4] suggested a recursive approach based on the so-called $k$-sum decomposition for $k \leq 3$, which has been shown to exist for this class by Wagner [41]. For graphs with bounded treewidth and a given tree decomposition, Wimer [43] has shown that a bottom-up approach based on dynamic programming will solve the maximum cut problem in linear time.

Vertex separators and $k$-connected components for small $k$ have been studied in many publications. For $k \leq 3$, $k$-connected components can be identified in linear time. While the decomposition of a graph into its biconnected components can be computed running a modified depth-first-search [40], the algorithm for decomposition into its triconnected components is more involved [22, 19]. Using Tarjan’s decomposition algorithm [40] and the data structure of SPQR-trees [39], the set of all vertex separators of sizes $k \leq 2$ can be enumerated efficiently. There are also theoretical approaches for identifying 4-connected components, however, they admit no clear definition [18]. Still, vertex-separators of size 3 can be found in $O(n^2)$ [27] and for general $k$ techniques based on maximum flow algorithms have been employed [13, 26]. Vertex separators have been made use of to speed up algorithms in the past: E.g. Iwata and Shigemura [24] employ a vertex separator-based pruning in a dynamic programming approach for Steiner trees. Hüffner et al. [23] developed reduction rules for signed graph balancing based on vertex separators. We will discuss the differences to our approach in Section 4.

The most recent and well-performing data reduction techniques for MaxCut from the literature will be covered in the next two sections. We refer to [1] for a general overview of recent progress in data reduction algorithms for problems in NP and P.

3 Edge Weight Based Reduction

One type of data reduction rules for MaxCut considers the edges (and their weights) with exactly one endpoint in a set of vertices $S$. We present them here with our new rule, complementing two existing ones. All rules of this type (implicitly) make use of the following observations.

\textbf{Observation 3.1.} Let $G = (V, E, w)$ be a simple undirected weighted graph. If for an edge $e = \{u, v\} \in E$ in $G$ we can prove the existence of a maximum cut $\delta$ with $e \not\in \delta$, the contraction of $u$ and $v$ is a valid data transformation (with $\beta = 0$).

In some cases, we find a proof for an edge $e \in E$ that there exists a maximum cut $\delta$ with $e \in \delta$. Here we can not make use of the above Observation 3.1 right away, but Fact 2 and Definition 4 in the work of Lange et al. [29] still allow to derive a data reduction rule. We restate their observations, for self-containedness:
Proposition 3.2. Let $G$ be an undirected weighted graph $G = (V, E, w)$ and $\delta^*$ any cut in $G$. Transforming $G$ into $H = (V, E, w')$, by negating all weights of edges in $\delta^*$ is a valid data reduction with $\beta = \sum_{e \in \delta^*} w_e$. If an edge $e$ is part of an optimal cut $\delta^*$ and also of $\delta^*$*, there exists an optimal cut $\delta_H$, which does not contain $e$.

With this, if a maximum cut $\delta$ with $\{u, v\} \in \delta$ is guaranteed to exist, we can either use $\delta^* = \delta(\{u\})$ or $\delta^* = \delta(\{v\})$ as the cut for the transformation from $G$ to $H$ and then make use of Observation 3.1 for a contraction in $H$. See Appendix A.1 for a proof.

Dominating Edges

Lange et al. [29] derived data reduction rules for edges with relatively high absolute weight:

Proposition 3.3 (Dominating Edge). Let $G = (V, E, w)$. If for any edge $e = \{u, v\} \in E$ and a subset $U \subset V$ with $e \in \delta(U)$ the inequality

$$|w_e| \geq \sum_{e \in \delta(U) \setminus \{e\}} |w_e|$$

holds, then there exists a cut $\delta$ with maximum value with $e \notin \delta$ if $w_e \leq 0$ and $e \in \delta$ if $w_e \geq 0$.

Proposition 3.3 naturally extends into a data reduction rule with Observation 3.1 and Proposition 3.2. For vertices with degree one or two, the condition is always true, therefore they can always be removed. Finding candidate sets $U$ for data reduction can be done via Gomory-Hu trees [17]. As already in previous work [29, 35], in our experiments we opt for the faster approach and only consider $U = \{u\}$ and $U = \{v\}$ for an edge $\{u, v\}$, resulting in $O(|E|)$ time.

Similar Vertices

For vertices with similar neighborhoods, one can identify cases in which both end up in the same / opposite partitions of an optimal solution, as Rehfeldt et al. [35] showed.

Proposition 3.4 (Similar Vertices). Let $G = (V, E, w)$. If two vertices $u, v \in V$ have the same neighborhood (excluding each other) $N(u) \setminus \{v\} = N(v) \setminus \{u\}$ and there exists an $\alpha \neq 0$ with $w_{uv} = \alpha w_{v'u'} \forall e, e' \in N(u) \setminus \{v\}$, then, if

- $\alpha > 0$ and $\{u, v\} \notin E$ or $w_{uv} \leq 0$, there is an optimal solution with $u$ and $v$ in same partition
- $\alpha < 0$ and $\{u, v\} \notin E$ or $w_{uv} \geq 0$, there is an optimal solution with $u$ and $v$ in different partitions

As also described in [35] candidates for this rule can be found quickly by making use of hashing techniques and we follow their suggestion in our implementation.

Triangles

For triangles in the input graph, the literature [29, 35] suggests two closely related data reduction rules. We complement them with a third and new one.

Proposition 3.5 (Triangles). Let the edges $\{v_1, v_2\}, \{v_1, v_3\}$ and $\{v_2, v_3\}$ form a triangle in $G$. Additionally, let $U_1 \subset V$ such that $\{\{v_1, v_2\}, \{v_1, v_3\}\} \subseteq \delta(U_1)$ and $U_2 \subset V$ such that $\{\{v_1, v_2\}, \{v_2, v_3\}\} \subseteq \delta(U_2)$. 

\[\alpha w_{v_1v_2} + \alpha w_{v_1v_3} + \alpha w_{v_2v_3} = \alpha \sum_{e \in E} w_e \]
1) *(introduced in [29]*) If the two inequalities
\[
-w_{v_1v_2} - w_{v_1v_3} \geq \sum_{e' \in \delta(U_1) \setminus \{(v_1, v_2), (v_1, v_3)\}} |w_{e'}|
\]
\[
-w_{v_1v_2} - w_{v_2v_3} \geq \sum_{e' \in \delta(U_2) \setminus \{(v_1, v_2), (v_2, v_3)\}} |w_{e'}|
\]
hold, there exists a cut \(\hat{\delta}\) with maximum value with \(\{v_1, v_2\} \notin \hat{\delta}\).

2) *(introduced in [35]*) If the two inequalities
\[
w_{v_1v_2} + w_{v_1v_3} \geq \sum_{e' \in \delta(U_1) \setminus \{(v_1, v_2), (v_1, v_3)\}} |w_{e'}|
\]
\[
w_{v_1v_2} - w_{v_2v_3} \geq \sum_{e' \in \delta(U_2) \setminus \{(v_1, v_2), (v_2, v_3)\}} |w_{e'}|
\]
hold, there exists a cut \(\hat{\delta}\) with maximum value with \(\{u, v\} \notin \hat{\delta}\).

3) *(new)* If the two inequalities
\[
-w_{v_1v_2} + w_{v_1v_3} \geq \sum_{e' \in \delta(U_1) \setminus \{(v_1, v_2), (v_1, v_3)\}} |w_{e'}|
\]
\[
-w_{v_1v_2} + w_{v_2v_3} \geq \sum_{e' \in \delta(U_2) \setminus \{(v_1, v_2), (v_2, v_3)\}} |w_{e'}|
\]
hold, there exists a cut \(\hat{\delta}\) with maximum value with \(\{v_1, v_2\} \notin \hat{\delta}\).

**Proof.** For the proof of 1) see [29]. For 2) [35] present a proof, but we suggest a more compact one based on Proposition 3.2, which also shows the correctness of our new implication, 3):

If we apply the technique from Proposition 3.2 to \(G\), by choosing \(\delta^* = \delta(v_1)\) for the transformation, resulting in \(G'\) and 1) holds for \(G'\), we see that there is maximum cut \(\hat{\delta}'\) in \(G'\) with \(\{v_1, v_2\} \notin \hat{\delta}'\) and therefore, because of Proposition 3.2, a maximum cut \(\hat{\delta}\) in \(G\) with \(\{v_1, v_2\} \notin \hat{\delta}\). But for 1) to hold in \(G'\), 2) needs to hold in \(G\) proving the correctness of 2). Following the same pattern, 3) can be derived from 2). Just choose \(\delta^* = \delta(v_2)\) for the transformation from \(G'\) to \(G''\) and 2) holds in \(G''\) if 3) holds in \(G\).

As for Proposition 3.3, Proposition 3.5 also extends into data reduction rules with Observation 3.1 and Proposition 3.2. We again follow [35] who consider \(\{v_1\}\) and \(\{v_2, v_3\}\) for \(U_1\) and \(\{v_2\}\) and \(\{v_1, v_3\}\) for \(U_2\) in their implementation.

### 4 Vertex Separator Based Reduction

Vertex separators of size one split the graph into its biconnected components, which can be solved independently from each other. Vertex separators of larger sizes also allow for preprocessing to be effective, as we will show in this section. Some of the data reduction rules for MAXCUT discussed in the literature already make implicit use of vertex separators. We introduce a generalized framework, show how and which new and existing rules can be derived through it, and formalize all cases for which the rules derived from our framework are valid.
The Vertex Separator Framework

We consider vertex separator-related data reduction rules, captured by the following definition. See Figure 1 for an example.

**Definition 4.1 (Data reduction rule type Separator).** For a vertex separator $S$ in $G = (V,E,w)$ separating $H \subset V$ from the rest of the graph, a rule is of type **Separator**, if it reduces $G$ to $G' = G[V \setminus H]$, effectively deleting all vertices in $H$ from $G$. All edges keep their original weight, apart from those in $G'[S]$.

Note: For simplicity we assume $G[S]$ to always be fully connected. If this is not the case, the missing edges can be added with a weight of zero (as this is a MaxCut-valid data transformation with $\beta = 0$). Edges in $G'[S]$ (may) get their weights updated, to encode the MaxCut values of $G[H \cup S]$ for all possible bipartitions of $S$. The goal is to ensure Condition 2.1 for every bipartition $P$ of vertices in $G'[S]$, i.e., the maximum cut value in $G'[S]$ needs to be the same (plus some constant $\beta$) as the one in $G[H \cup S]$ when fixing the partition of the vertices in $P$.

Hüffner et al. [23] developed a concept for the signed graph balancing problem, which is related to Definition 4.1. In their work, a set of vertices, only connected to the graph by a separator, gets replaced by a gadget. This introduces new vertices and edges to the graph. Our rules of type **Separator** only remove and never add vertices. In the following we provide a sufficient condition, that allows us to safely apply data reduction rules based on vertex separators.

**Theorem 4.2.** Let $S$ be a vertex separator in $G = (V,E,w)$ separating $H \subset V$ from the rest of the graph, and $P_S$ be the set of all possible bipartitions of vertices in $S$. Then a data reduction rule of type **Separator** is valid for $G$ if the system of equations

$$\sum_{e \in \delta_G(S)} (w_e + \gamma_e) + \beta = \Delta(G[H \cup S], P) \quad \forall P \in P_S$$

has a feasible solution for the variables $\gamma_e$ and $\beta$.  

\textbf{Figure 1} Example of a reduction rule of type vertex separator. (a) Original graph $G$ with vertex separator $S = \{a, b, c\}$ separating $H$ from the rest of the graph. (b) The resulting reduced graph $G'$ with adapted edge weights, i.e., $H$ has been deleted from the remaining graph clearly reducing the size of the instance.
4:8 Separator Based Data Reduction for the Maximum Cut Problem

Proof. Let \( G' = G[V \setminus H] \) and all edges in \( G'[S] \) get their weight updated by adding the corresponding \( \gamma_e \). This transformation fulfills Condition 2.1 and is a valid data reduction as:

1) For any bipartition \( (U', U') \) in \( G' \) with MAXCUT value \( c' \) we can find a bipartition \( (U, \bar{U}) \) in \( G \) with a cut value of \( c \geq c' + \beta \), because by construction of \( G' \) combining \( (U', U') \) with the optimal partitioning of the vertices in \( V(G) \setminus V(G') \) we get one for \( G \) with a cut value of exactly \( c' + \beta \).

2) For any bipartition \( P = (U, \bar{U}) \) in \( G \) with MAXCUT value \( c \) we can make sure we can map it to a bipartition \( (U', U') \) in \( G' \) with a cut value \( c' + \beta \geq c \), as again by construction of \( G' \) the bipartition resulting from removing all vertices from \( P \) which are not in \( G' \) results in a bipartition whose cut value in \( G' \) is at least as high as the one of \( P \) in \( G \). ▶

Small Separators

We now explicitly consider small vertex separators of size \( k = 2 \) and \( k = 3 \) and derive concrete and valid rules of type \textsc{separator}. These two rules are known in theory, where they have been introduced in the context of polynomial algorithms for \( K_5 \) and \( K_{3,3} \) minor-free graphs \([4, 10]\). To the best of our knowledge, they have not been considered for general graphs before and also have not been used in practice up to now. We start with separators of size 2:

**Corollary 4.3.** Let \( G = (V, E, w) \) be a graph and \( S = \{a, b\} \) a 2-separator in \( G \) separating \( H \subset V \) from the rest of the graph. For more compact notation define \( \tilde{H} := H \cup S \). Then the data reduction rule of type \textsc{separator} is valid with the following values:

\[
\beta = \Delta(G[\tilde{H}], \{(a, b), \emptyset\}) \quad \text{and} \quad \gamma_{ab} = \Delta(G[\tilde{H}], \{(a), \{b\}\}) - \Delta(G[\tilde{H}], \{(a, b), \emptyset\}) - w_{ab}.
\]

Proof. There are two possible bipartitions for vertices in \( S \) and \( G[S] \) contains one edge \( (e = \{a, b\}) \). Therefore the equation system of Theorem 4.2 gives two equations and two variables.

\[
\beta = \Delta(G[\tilde{H}], \{(a, b) : \emptyset\}) \quad \text{and} \quad w_{ab} + \gamma_{ab} + \beta = \Delta(G[\tilde{H}], \{(a) : \{b\}\})
\]

Solving the system for \( \beta \) and \( \gamma_{ab} \) yields the stated equations. ▶

This general concept for vertex separators of size 2 covers some existing rules, like rule 2 and 6 of \([14]\) as special cases. Vertex separators of size 3 allow for a generalized data reduction rule in a similar way:

**Corollary 4.4.** Let \( G = (V, E) \) be a graph and \( S = \{a, b, c\} \) a 3-separator in \( G \) separating \( H \) from the rest of the graph. For more compact notation define \( \tilde{H} := H \cup S \). Then the data reduction rule of type \textsc{separator} is valid with the following constant offset \( \beta \) and values for \( \gamma \):

\[
c_0 := \Delta(G[\tilde{H}], \{(a, b, c) : \emptyset\}) \quad \text{and} \quad \beta = c_0
\]

\[
c_1 := \Delta(G[\tilde{H}], \{(a, b) : \{c\}\}) \quad \text{and} \quad \gamma_{ac} = 1/2 \cdot (c_1 + c_3 - c_0 - c_2) - w_{ac}
\]

\[
c_2 := \Delta(G[\tilde{H}], \{(a, c) : \{b\}\}) \quad \text{and} \quad \gamma_{ab} = 1/2 \cdot (c_2 + c_3 - c_0 - c_1) - w_{ab}
\]

\[
c_3 := \Delta(G[\tilde{H}], \{(b, c) : \{a\}\}) \quad \text{and} \quad \gamma_{bc} = 1/2 \cdot (c_1 + c_2 - c_0 - c_3) - w_{bc}
\]

Proof. There are four possible bipartitions for vertices in \( S \) and \( G[S] \) contains three edges. Therefore the equation system of Theorem 4.2 gives four equations and four variables. The constants \( c_0, \ldots, c_3 \), introduced for better readability, represent the right hand sides of the equation system. Solving this system for the constant offset \( \beta \) and the edge weights gives the presented equations. ▶
Separators of Arbitrary Size

For separators larger than three the above techniques only work in specific cases, as the number of potential bipartitions outgrows the number of possible edges in \( G[S] \), resulting in an overdetermined system of equations. For a 4-separator we already get eight partitions (resulting in eight equations), but only seven variables (six for edges and \( \beta \)). Nevertheless for graphs with unit weights special rules have been described, for which their proof can be simplified through the lens of our framework and a new and more general one can be deduced. This new rule generalizes rule 1 and extends 5 and 7 from Ferizovic et al. [14] into a less restrictive and therefore more often applicable rule.

**Proposition 4.5 (Same Neighborhood Clique).** Let \( G = (V, E, w) \). If for a clique \( C \subset V \) in \( G \), for which \( w_e = 1 \) for all edges \( e \) incident to at least one vertex in \( C \), \( |C| + 1 \geq |N(C)| \geq 1 \) and \( N(C) = N(u) \setminus C \ \forall u \in C \), removing vertices in \( C \) from the graph and updating weights of edges between vertices in \( N(C) \) is a valid data reduction of type \textit{Separator}. The constant offset is \( \beta = \left[ \frac{|N(C)| + |C|}{2} \right] \). All weights of edges in \( G[N(C)] \) get reduced by 1.

**Proof.** The vertex set \( S := N(C) \) forms a vertex separator of \( G \) with \( C \) on one side. Recall, we assume w.l.o.g. \( G[S] \) to be fully connected. Consider the equations from Theorem 4.2: Subtracting the weight of edges contributing to the value of the cut from both sides, leaves \( \sum_{e \in E(S \cup C)} \gamma_e + \beta \) as the left-hand side. The resulting right-hand side \( \Delta(G[S \cup C], P) - \sum_{e \in E(S \cup C)} \delta_{G[S \cup C]}(P)(w_e) \) can be interpreted as the value of the maximum cut in \( G' \): \( \Delta(G', P) \), where \( G' = G[S \cup C] \), except all edges incident to two vertices in \( S \) have a weight of 0. Because of \( |S| \leq |C| + 1 \), no matter how vertices in \( G'[S] \) get partitioned, \( \Delta(G', P) \) can always be maximized by partitioning vertices in \( S \cup C \) into two partitions, whose sizes differ by at most one. Therefore, for the partial bipartition \( P = (S, \emptyset) \), the value of \( \Delta(G', P) \) is \( \beta = \left[ \frac{|S| + |C|}{2} \right] \). For general \( P \) this value has to be reduced by the number of edges cut in \( G'[S] \), as they have a weight of 0 in \( G' \). If \( G[N(C)] \) is fully connected with weight 1 edges, then \( G[C \cup N(C)] \) forms a clique and thus Proposition 4.5 leads to the same reduction as rule 1 of [14]. Rule 5 and 7 of their work have the same type of clique as their nucleus, but only allow the removal of vertices from \( C \) until \( |C| = |N(C)| \). Candidates for the rule resulting from Proposition 4.5 can be found fast exactly as described by Ferizovic et al. [14].

**Misc.** Of the seven rules of Ferizovic et al. [14], two (rule 3 and 4) do not exactly fit into our \textit{Separator} framework. Their rule 4 removes an edge from a clique, which creates candidates for Proposition 3.4 and their rule 3 adds an edge to the graph, which might lead to candidates for Proposition 4.5, hence we restate them here:

**Proposition 4.6 (Near Clique -).** Let \( G = (V, E, w) \) and \( C \subset V \) be a clique where all edges connecting two vertices in \( C \) have weight 1. Define \( C_{in} \subset C \) as the set of vertices with neighbors in \( C \) only. If \( |C| \) is odd or \( |C_{in}| > 2 \), removing one edge connecting vertices in \( C_{in} \) is a valid data transformation with \( \beta = 0 \).

**Proposition 4.7 (Near Clique +).** Let \( G = (V, E, w) \) and \( C \subset V \) be a subgraph missing only one edge \( e = \{u, v\} \), where all edges connecting two vertices in \( C \) have weight 1. Define \( C_{in} \subset C \) as the set of vertices with neighbors in \( C \) only. If \( u, v \in C_{in} \) and \( |C| \) is odd or \( |C_{in}| > 2 \), adding \( e \) is a valid data transformation with \( \beta = 0 \).
Algorithmic Framework

We now present the algorithm we derived from the existing and our new data reduction rules presented in the preceding sections. The algorithm consists of two components, decomposition and data reduction.

Decomposition

Let $G$ be the input graph for our algorithm. We keep a queue of tuples $(g, l)$. The first element of the tuple is a subgraph of $G$ and the second is a lower bound on the size of any vertex separator in $g$. In the beginning, we insert the tuple $(G, 0)$ into the queue. We pop elements $(g, l)$ one by one from the queue. We first check if $\text{MaxCut}$ is easy to solve on $g$ (for example because $g$ is small or has a special structure) and if so we solve $g$ immediately. Otherwise if $l \leq 1$ we first check if $g$ is (bi-)connected. If not we compute the (bi-)connected components and add them to the queue with the lower bound on vertex separators of $l + 1$. If neither of these steps worked we attempt to reduce the graph as much as possible using the core data reduction algorithm.

In more detail, the outer / decomposition part of our algorithm removes a tuple $(g, l)$ from the queue and checks in order:

- If $g$ has $\leq k$ vertices, we calculate $\Delta(g)$ by complete enumeration right away. If all edges of $g$ have weight 1, we make use of a linear time algorithm (see Appendix A.2 for details), which either outputs an optimal $\text{MaxCut}$ value or reports that the special structure it is designed for is not present. In the latter case, we just continue.
- If $l = 0$ we test whether or not the graph is connected and add each connected component $g_i$ as the tuple $(g_i, 1)$ to the queue.
- If $l = 1$ we test whether or not the graph is biconnected and add each biconnected component $g_i$ to the queue, as the tuple $(g_i, 2)$.
- If the graph is at least biconnected, we apply the core data reduction algorithm.

Data Reduction

The core data reduction algorithm is based on all data reduction rules presented earlier:

- If the graph has unit weights we first reduce based on the data reduction rules of Ferizovic et al. [14] designed for this special case and our new one (Proposition 4.5).
- For the resulting graph we apply: The dominating-edge rule from Proposition 3.3, the rule for similar vertices from Proposition 3.4, and all three triangle rules of Proposition 3.5, including our new one. Additionally, we use the SEPARATOR rule for vertex separators of size 3 (Corollary 4.4) by deleting vertices of degree three and updating the edge weights between their neighbors.
- Finally we turn to our rule of type SEPARATOR for 2-separators: We calculate the SPQR-tree of the remaining graph to remove small subgraphs based on Corollary 4.3. Leaves of the SPQR-tree can be contracted recursively into the edge of the 2-separator, by calculating the solutions for two $\text{MaxCut}$ problems on the leaf. For finding exact solutions we again use our enumeration algorithm and therefore only apply this for SPQR-tree leaves of size $\leq k$.

Note: The restriction to only process small ($\leq k$) leaves of the SPQR-tree is necessary, to guarantee the data reduction is feasible (has polynomial runtime in the input size). Between any of the above steps, we make sure the graph is still (bi-)connected. If it loses its (bi-)connectivity, we add the (bi-)connected components to the queue and break. The
procedure could be augmented to search for general 3-separators, but the quadratic running
time of finding 3-separators [26] might not be suitable for big graphs, so we restrain the
algorithm to vertices of degree three here.

To make steps (i) and (ii) efficient, we employ queues and markers to keep track of vertices
for which the neighborhood has changed and who might be candidates for one of the data
reduction rules. Similar techniques have been suggested by e.g. Ferizovic et al. [14].

6 Computational Experiments

We extensively evaluate the practical impact of our new techniques in multiple ways and
compare them against the state-of-the-art. First, we showcase the effectiveness of our rules,
measured by the number of vertices and edges removed from the original graph. Second, we
highlight the usefulness of our new rules for solving MaxCut in practice. We show that
our rules not only result in smaller graphs than the current state-of-the-art, but also help
to further speed up an exact state-of-the-art solver and improve the solution quality of a
state-of-the-art heuristic. Our rules are therefore not only effective but also efficient. Finally,
we perform an ablation study, investigating the influence each of our new components has.

Setup and Solvers. The system we employed for the experiments is equipped with an AMD
Ryzen Threadripper 3960X CPU, 128GB of RAM, and has Ubuntu 22.04. as its operating
system. Our code is written in C++20, compiled with GCC 11.3. and the “-O3” flag. We
make use of multiple open-source libraries, especially NetworKit [38] and OGDF [9] for graph
algorithms and simexpal [2] for our experiments. On this system our naive enumeration
algorithm takes clearly below one second to calculate the MaxCut value for graphs up to 21
vertices, therefore we set \( k \) to 21 in our algorithmic framework (see Section 5).

We use Gurobi 10 for experiments with an exact solver, as recent benchmarks by Hans
Mittelmann [33] show Gurobi to be the solver, solving the second most instances to optimality
before reaching the timelimit. The only solver with a better result is the non-publicly available
QuBowl [35]. To allow Gurobi to efficiently solve MaxCut, we convert the problem to an
unconstrained binary quadratic problem, which is a well-known transformation (see e.g. [6]).
We restrict Gurobi to the use of one thread and set the MIP-Gap to \( 10^{-6} \). To make the
comparison more interesting, we also give Gurobi a hint on the reducibility of the instances,
by setting the presolve parameter to \textit{aggressive}.

For experiments with a state-of-the-art heuristic we use the implementation of Dunning
et al. [12] of the algorithm of Burer et al. [7]. This algorithm did not only perform very
well in the study of Dunning et al. [12], but is also employed in the state-of-the-art solver
QuBowl [35].

For all our benchmarks, we run the same experiment with multiple different random
seeds. This is to compensate for performance variability [31]. The seed is used to shuffle
and relabel all vertices, as this change can lead to different orders of data reduction rule
applications and therefore the resulting graphs differ. In experiments involving Gurobi, the
seed is also passed to the solver.

Instances. We collected all instances from the most recent purely data reduction dedicated
study we are aware of by Ferizovic et al. [14]. We also added publicly available instances
Rehfeldt et al. [35] report results of their preprocessing on, for which preprocessing is effective,
but the remaining graph does not become extremely small. The resulting set has no instances
with \( 10000 \leq |V| \leq 200000 \). We fill this gap with graphs of the same structure and from the
Table 1 Characteristics of the benchmark instances considered in our study. The columns \(d\) and \(\bar{d}\) capture the min and max degree, \(w\) and \(\bar{w}\) show the min and max edge weight.

| set     | instance         | \(|V|\) | \(|E|\) | \(d\) | \(\bar{d}\) | \(w\) | \(\bar{w}\) |
|---------|------------------|--------|-------|-------|-----------|-------|-----------|
| easy    | soc-firm-hi-tech | 33     | 91    | 1     | 16        | 1     | 1         |
| easy    | g001207          | 84     | 149   | 1     | 5         | 1     | 100000    |
| easy    | g0009081         | 110    | 188   | 2     | 6         | 1     | 100000    |
| easy    | ENZYMES_g295     | 123    | 139   | 1     | 5         | 1     | 1         |
| easy    | g000292          | 212    | 381   | 2     | 4         | 5     | 13        |
| easy    | g000302          | 317    | 476   | 1     | 5         | 1     | 13        |
| easy    | ca-netscience    | 379    | 914   | 1     | 34        | 1     | 1         |
| easy    | bio-diseasome    | 516    | 1188  | 1     | 50        | 1     | 1         |
| easy    | rt-twitter-copen | 761    | 1029  | 1     | 37        | 1     | 1         |
| easy    | g001918          | 777    | 1239  | 1     | 4         | 5     | 13        |
| easy    | imgseg_271031    | 900    | 1027  | 1     | 518       | 93899 | 059 285968046836 |
| easy    | road-euroroad    | 1174   | 1147  | 1     | 10        | 1     | 1         |
| easy    | imgseg_35058     | 1274   | 1806  | 1     | 587       | -55510850 | 118 112271093673 |
| easy    | bio-yeast        | 1458   | 1948  | 1     | 56        | 1     | 1         |
| easy    | imgseg_106025    | 1565   | 2629  | 1     | 902       | 93891365 136834528589 |
| easy    | ca-CSphd         | 1882   | 1470  | 1     | 46        | 1     | 1         |
| easy    | ego-facebook     | 2888   | 2981  | 1     | 769       | 1     | 1         |
| easy    | imgseg_105019    | 3548   | 4325  | 1     | 2753      | 1096232 | 218 236593516427 |
| easy    | imgseg_374020    | 5735   | 8722  | 1     | 2213      | -466398208299 407957172555 |
| medium  | web-google      | 1299   | 2773  | 1     | 59        | 1     | 1         |
| medium  | inf-power        | 4941   | 6594  | 1     | 19        | 1     | 1         |
| medium  | ca-Erdos992      | 5094   | 7515  | 1     | 61        | 1     | 1         |
| medium  | g000677          | 17127  | 27352 | 1     | 4         | 1     | 126       |
| medium  | g001075          | 27019  | 39407 | 1     | 4         | 1     | 228668    |
| medium  | imgseg_147062    | 28552  | 65453 | 1     | 925       | -1567963186 37500110 |
| medium  | g000087          | 38418  | 71657 | 2     | 4         | 1     | 198       |
| medium  | road-luxembourg-osm | 114599 | 119666 | 1 | 6        | 1     | 1         |
| big     | web-Stanford     | 281903 | 1992636 | 1 | 38625 | 1     | 1         |
| big     | ca-MathSciNet    | 332689 | 820644 | 1 | 496    | 1     | 1         |
| big     | web-it-2004      | 509338 | 7178413 | 1 | 469   | 1     | 1         |
| big     | ca-coauthors-dblp | 540486 | 15245729 | 1 | 3299  | 1     | 1         |
| big     | ca-IMDB          | 896305 | 3782446 | 1 | 1590  | 1     | 1         |
| big     | inf-road_central | 140818 | 16933413 | 1 | 8     | 1     | 1         |
| torus   | t2g10            | 100    | 200   | 4     | 4        | -294541 300104 |
| torus   | t2g15            | 225    | 450   | 4     | 4        | -294541 375001 |
| torus   | t2g20            | 400    | 800   | 4     | 4        | -294541 375001 |
| torus   | t2g5             | 125    | 375   | 6     | 6        | -294541 290339 |
| torus   | t2g6             | 216    | 648   | 6     | 6        | -294541 375001 |
| torus   | t2g7             | 343    | 1029  | 6     | 6        | -298103 375001 |

*For every type of torus graph, there are three instances of the same size. Values are aggregated.

same source as Ferizovic et al. [14]. We group the instances in four sets, based on their type, size, and difficulty. Instances from the “easy” set can be solved in (sometimes significantly) less than ten seconds by the state-of-the-art exact solver Gurobi and are mainly included for comparability to earlier studies (they form the “medium” set of [14] and stem from [36] and [12]). Instances from our set “medium” require roughly ten seconds or more to solve by Gurobi and contain two of the “hard” instances of [14] as well as instances from the Network Repository [36] and the MQLib [12]. Large instances, too challenging for Gurobi as an exact solver, get assigned to the “big” set. The torus instances [42] from statistical physics get their own category, as these graphs have grid structure and therefore differ heavily from the real-world graphs in other sets. Table 1 summarises our set of benchmark instances.

6.1 Effectiveness and Efficiency

We start off by comparing the effectiveness of our data reduction algorithm with the current state-of-the-art in Table 2. In the following, we choose our implementation of the state-of-the-art algorithm of Rehfeldt et al. [35] as the baseline, because: 1) we wanted to rule out
side effects, resulting from applying rules in different orders and other implementation details and 2) the only publicly available data reduction code for MaxCut by Ferizovic et al. [14] is not competitive with our state-of-the-art baseline (see Appendix A.3 for details).

### Table 2
Average effectiveness of our implementation of the state-of-the-art (sota) preprocessing of [35] and our new algorithm. Percentage of remaining vertices / edges, as well as average runtime in seconds (pr[s]). The improvement columns show, how many vertices / edges relative to the sota algorithm could be removed additionally.

<table>
<thead>
<tr>
<th>set</th>
<th>sota</th>
<th>our</th>
<th>improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>easy</td>
<td>10.20</td>
<td>13.73</td>
<td>0.02</td>
</tr>
<tr>
<td>medium</td>
<td>19.64</td>
<td>27.16</td>
<td>0.09</td>
</tr>
<tr>
<td>big</td>
<td>37.41</td>
<td>53.31</td>
<td>0.09</td>
</tr>
<tr>
<td>torus</td>
<td>78.32</td>
<td>85.02</td>
<td>0.00</td>
</tr>
</tbody>
</table>

### Table 3
Runtime comparison on the “medium” instances. The runtimes refer to pure Gurobi (gurobi) and to Gurobi with our implementation of the state-of-the-art (sota) preprocessing and our preprocessing. tr[s] is the total runtime it took, to calculate the optimal solution. For sota and our, the remaining columns report the percentage of vertices / edges left after the preprocessing and pr[s] is the reduction runtime only. The speedup (spd) is sota total runtime divided by our total runtime. All values are averages over 5 seeds per instance. Timeout refers to more than 3600s.

<table>
<thead>
<tr>
<th>instance</th>
<th>gurobi</th>
<th>sota</th>
<th>our</th>
<th>spd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>tr[s]</td>
<td>V [%]</td>
<td>E [%]</td>
<td>pr[s]</td>
</tr>
<tr>
<td>web-google</td>
<td>9.08</td>
<td>5.60</td>
<td>11.18</td>
<td>0.08</td>
</tr>
<tr>
<td>inf-power</td>
<td>9.25</td>
<td>16.15</td>
<td>23.47</td>
<td>0.02</td>
</tr>
<tr>
<td>ca-Erdős992</td>
<td>1152.01</td>
<td>14.92</td>
<td>36.43</td>
<td>0.01</td>
</tr>
<tr>
<td>g000677</td>
<td>58.07</td>
<td>22.04</td>
<td>28.94</td>
<td>0.03</td>
</tr>
<tr>
<td>g001075</td>
<td>45.08</td>
<td>11.38</td>
<td>15.93</td>
<td>0.04</td>
</tr>
<tr>
<td>imgseg_147062</td>
<td>2757.79</td>
<td>49.65</td>
<td>56.78</td>
<td>0.22</td>
</tr>
<tr>
<td>g000087</td>
<td>timeout</td>
<td>32.99</td>
<td>38.08</td>
<td>0.05</td>
</tr>
<tr>
<td>road-luxembourg-osm</td>
<td>40.19</td>
<td>4.39</td>
<td>6.51</td>
<td>0.27</td>
</tr>
</tbody>
</table>

### Table 4
Solution value comparison on the “big” instances. The heuristic (burer) of Burer et al. [7] was run for 1800s. When paired with the state-of-the-art (sota) or our preprocessing, the runtime was reduced by the time the preprocessing took (pr[s]). The bv column shows the best value found when no preprocessing is employed. bvi is the absolute improvement over pure burer. The remaining columns report the percentage of vertices / edges left after the preprocessing. All values are averages over 5 seeds per instance.

<table>
<thead>
<tr>
<th>instance</th>
<th>burer</th>
<th>sota</th>
<th>our</th>
<th>bvi</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bv</td>
<td>V [%]</td>
<td>E [%]</td>
<td>pr[s]</td>
</tr>
<tr>
<td>web-standford</td>
<td>1584791</td>
<td>52.15</td>
<td>61.47</td>
<td>15.30</td>
</tr>
<tr>
<td>ca-MathSciNet</td>
<td>600010</td>
<td>30.82</td>
<td>52.58</td>
<td>2.53</td>
</tr>
<tr>
<td>web-ii-2004</td>
<td>4052029</td>
<td>4.11</td>
<td>4.94</td>
<td>2.56</td>
</tr>
<tr>
<td>ca-coauthors-dblp</td>
<td>8219970</td>
<td>70.31</td>
<td>85.77</td>
<td>35.69</td>
</tr>
<tr>
<td>ca-IMDB</td>
<td>3361135</td>
<td>46.74</td>
<td>87.18</td>
<td>10.07</td>
</tr>
<tr>
<td>inf-road_central</td>
<td>15349816</td>
<td>29.30</td>
<td>27.90</td>
<td>172.12</td>
</tr>
</tbody>
</table>
The effectiveness of our proposed algorithmic framework can clearly be seen, as we improve upon the state-of-the-art on every instance set. For the “easy” set the most significant improvement can be observed; On average only $3.93\%$ of edges remain after applying our data reduction algorithm. For 15 out of the 19 instances, our algorithm fully solves the input to optimality, resulting in an empty transformed graph. This is only the case for 8 instances with the sota algorithm. The torus set profits the least from preprocessing, because of the special structure of those graphs. Still, applying our rules does not increase the runtime and helps to remove some additional vertices / edges.

Next, we show the effect of our preprocessing on the runtime of an exact state-of-the-art solver. Instances in the “easy” set can be solved extremely fast by state-of-the-art solvers like Gurobi. They require 0.32s with our implementation of the state-of-the-art preprocessing and 0.29s with our preprocessing on average. For the “torus” set, the overall speedup over the state-of-the-art is about 1.5\%, which is to be expected considering the small difference in effectiveness. The “big” instances seem too challenging for exact solvers like Gurobi, hence we will consider this set separately.

Table 3 shows the results for the “medium” set. For all instances Gurobi is faster when external preprocessing is employed and the total solving time is the smallest if our algorithm reduces the input. Directly comparing the total runtime of the state-of-the-art approach and our algorithm, shows speedups of up to one order of magnitude: For road-luxembourg-osm the solving time decreases from 23.14s to as little as 1.75s, a speedup of 13x. Overall the additional time required by our new rules, easily makes up for it, when solving real-world MaxCut instances to optimality. To investigate the efficiency on the “big” set, we compare the best solution values the heuristic of [7] finds on these graphs with a timelimit of 1800 seconds. Table 4 shows the results. Again, our preprocessing strictly improves over the state-of-the-art. For every instance, the additional time spent in our preprocessing pays off. Even in the case of ca-coauthors-dblp, where the preprocessing is slower by quite a bit (leading to a reduced runtime of the heuristic in our setting), the best solution found is on average better by 169. Also, relatively small differences in objective values (e.g. for web-it-2004) can matter greatly, if the instances get within reach of exact solvers.

6.2 Ablation study

As our preprocessing algorithm turns out quite powerful, we also investigate the influence of each component in more detail. Although our data reduction algorithm is useful for exact and heuristic solving, we restrict the analysis in the following to the exact solving of the “medium” set.

Cliquets and Triangles. The rule for cliques introduced in Proposition 4.5 is relevant for graphs, where all edges have the same positive weight. For these graphs, we experimented with turning off our rule and reverting to the version of Ferizovic et al. [14], as well as not applying rules for cliques of this type altogether. Similar to Ferizovic et al. in their experimental study we did not notice big differences in the sizes of the resulting graphs in general. Nevertheless, for web-it-2004 the setting has a significant impact. Without our generalized rule, 4.9% of all edges remained after applying our full data reduction algorithm. If we also included our new clique rule, only 3.7% of all edges remained, a 24% improvement.

The new rule for triangles from Proposition 3.5 has a noticeable impact on the graph size and the solving time of Gurobi, as shown in Figure 2. Making use of all three rules for triangles helps to reduce 6 out of the 8 graphs even further (web-google is fully reduced anyways and for ca-Erdos992 triangles have no effect). Especially for the weighted g00 and
imgseg instances, the rules tend to remove many edges. Also, including our new rule for triangles ("all-tri") always improves over deactivating it individually ("wo3-tri"). Sometimes the benefit seems small, but we clearly observed diminishing returns for all triangle rules. When activating triangle rules, independent of the order, every additional rule yields less benefit. As it also requires very little code to implement an additional triangle rule, if one is there already, we strongly suggest implementing all of them (including our new one).

**Figure 2** Effectiveness and efficiency of our full algorithm, with certain triangle-related rules turned on / off on the “medium” set. The variant with all triangle rules deactivated is no-tri and all-tri has all rules activated. In wo3-tri only our new rule is deactivated. Left: Relative number of remaining edges. Right: Total runtime in seconds (data reduction + Gurobi, logarithmic scaling). All values are averaged over five seeds per instance.

**Vertex Separator.** We now turn to our rules for 2- and 3-separators. To investigate their impact, we compare three settings of our algorithm in Figure 3. While most of the time the use of 2-separators only slightly reduces the size of the graph, also considering 3-separators has an impact on all graphs. We also see the total runtime (preprocessing + exact solving via Gurobi) does not always improve when 2-separators are considered; the extra effort for enumerating them is not always worth it. Our rule for 3-separators on the other hand, is not only extremely fast and effective but also attributes for much of the speedups over the state-of-the-art seen earlier in Table 2. E.g. for g000677 the total solving time decreases significantly, from about 24s to 8s, when 3-separators are also considered.

**7 Conclusion**

Our new vertex separator-based data reduction framework for MaxCut allows for the derivation of fast rules for separators of size 2 and 3 and additionally covers some known rules from the literature. The new separator-based rules and our new rule for triangles prove to be highly effective and efficient when paired with well-performing techniques from the literature and tested on established benchmarks. Even when purely employed as a preprocessing, our data reduction algorithm already fully solves 15 out of 19 instances from
Figure 3 Effectiveness and efficiency of different settings of our algorithm: The full algorithm including rules for 2- and 3-separators is con3. The version of our algorithm, where only the rule for 3-separators is turned off is con2. For con1, the data reduction making use of 2-separators via the SPQR-tree decomposition is disabled as well. Left: Relative number of remaining edges. Right: Total runtime in seconds (data reduction + Gurobi, logarithmic scaling) for each instance of the “medium” set. All values are averaged over five seeds per instance.

References


J. Charfreitag, C. Dahn, M. Kaibel, P. Mayer, P. Mutzel, and L. Schürmann
Separator Based Data Reduction for the Maximum Cut Problem


A Appendix

A.1 Proof of Proposition 3.2

Proposition 3.2. Let $G$ be an undirected weighted graph $G = (V, E, w)$ and $\delta^*$ any cut in $G$. Transforming $G$ into $H = (V, E, w')$, by negating all weights of edges in $\delta^*$ is a valid data reduction with $\beta = \sum_{e \in \delta^*} w_e$. If an edge $e$ is part of an optimal cut $\delta_G$ and also of $\delta^*$, there exists an optimal cut $\delta^*$, which does not contain $e$.

Proof. We first show, why the transformation from $G = (V, E, w)$ to $H = (V, E, w')$ with $\beta = \sum_{e \in \delta^*} w_e$ is a valid data transformation: As the only difference between $G$ and $H$ are edge weights, any cut in $G$ is a cut in $H$ and vice versa. The symmetric difference between all cuts in $G$ and $\delta^*$ is a bijection from cuts in $G$ to cuts in $H$. This bijection maps any cut with value $c$ in $G$ to one in $H$ with the exact same value (because of the definition of $\beta$). Therefore $\Delta(G) = \Delta(H) + \beta$ and the transformation is valid. Let $e$ be any edge in $G$ part of an optimal cut $\delta_G$ and also present in $\delta^*$. By transforming $G$ into $H$ with $\delta^*$, the symmetric difference of $\delta_G$ and $\delta^*$ gives a cut with optimal value in $H$, which does not contain $e$.

A.2 Solving certain unit weight graphs in linear time

In the decomposition part of our algorithm (see Section 5) we detect special graph structures, allowing for a linear-time algorithm for MAXCUT. The following algorithm calculates $\Delta(G)$ for graphs $G$ from two classes of graphs, for which all edges have the same positive weight, in time $O(n + m)$. The basic idea for the second case was described by Arbib [3].

1) Bipartite graphs and 2) graphs containing a $K_{\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil}$ subgraph.

Proof. A check if a graph is bipartite can be implemented in $O(n + m)$ and using the bipartition as the MAXCUT solution all edges are included in the cut, clearly making this the optimal solution.

If the graph is not bipartite we check if $m \geq \lfloor \frac{n}{2} \rfloor \cdot \lceil \frac{n}{2} \rceil$. If not the graph can not have a $K_{\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil}$ subgraph. Otherwise, we check for the existence of a $K_{\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil}$ subgraph by using the following observation about the connected components of the complement graph $G$: there is a $K_{\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil}$ subgraph in $G$ if there is a bipartition $(V_1, V_2)$ of $V$ with $|V_1| = \lfloor \frac{n}{2} \rfloor, |V_2| = \lceil \frac{n}{2} \rceil$ such that in $G$ there is no edge connecting vertices from $V_1$ to vertices in $V_2$. We can check for the existence of such a bipartition, by solving the SUBSETSUM Problem. The input is a list of the sizes of every connected component in $G$. We use the fact that SUBSETSUM reduces to KNAPSACK by making one item per given number $x_i$, which has $w_i = p_i = x_i$. Then we can use the dynamic programming algorithm for KNAPSACK by [11], which runs in time $O(nC)$ where $C$ is the capacity. As there are at most $n$ inputs for the SUBSETSUM problem and our target sum is $n$ this yields an $O(n^2)$ algorithm to check if a graph has a $K_{\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil}$ subgraph. As this case is only relevant, when $m \in \Omega(n^2)$, the whole algorithm runs in $O(n + m)$.

A.3 Additional Experiments

In our experiments in Section 6 we benchmark our new algorithm against our implementation of the state-of-the-art. To show the validity of this decision, we compare the preprocessing effectiveness of our implementation of the state-of-the-art and our new algorithm, with the
publicly available code\textsuperscript{1} of Ferizovic et al.\cite{14} in Table 5. All experiments were conducted on the same machine (see Section 6). Because it was developed in parallel / earlier, the implementation of Ferizovic et al.\cite{14} does not incorporate the rules of Lange et al.\cite{29} and Rehfeldt et al.\cite{35}. Hence, as expected, our state-of-the-art implementation (sometimes significantly) outperforms the older implementation in preprocessing effectiveness. Please note: Some results for FHLMSS reported here deviate from those in \cite{14} for two main reasons. For the “imgseg” instances, their paper mentions a scaling from floating point to integer via multiplication with 10e6. In the publicly available code, values are scaled with 10e5. We opted for 10e6, for higher precision. For the “g00” instances, we could only reproduce the results of Ferizovic et al.\cite{14} if unweighted versions of the original graphs are benchmarked. We use the original, weighted versions of all instances of type “g00”, as this was also the case in other studies \cite{8, 35}. For some rules, the implementation of Ferizovic et al.\cite{14} uses gadgets, to transform weighted input graphs into unweighted ones. As a result, large edge weights may lead to excessive RAM usage (in our experiments more than 128 GB) and we were not able to collect results for the “torus” and some “g00” graphs. For these, our state-of-the-art implementation did not require more than 27 MB of main memory.

\textsuperscript{1} We work with a public fork of the original code (https://github.com/Amtrix/fpt-max-cut), which offers some quality of life changes like a simplified compile step: https://github.com/CharJon/fpt-max-cut
**Table 5** Effectiveness comparison of three different preprocessing implementations: Ferizovic et al. [14] (**FHLMSS**), our baseline implementation of Rehfeld et al. [35] state-of-the-art (**sota**) preprocessing and our new algorithm. Percentage of remaining vertices ($|V|\%$) and edges ($|E|\%$), as well as runtime in seconds (**pr**[s]). The **na** entries indicate the implementation of **FHLMSS** required more RAM than available on the system (128GB) and therefore the data is not available.

<table>
<thead>
<tr>
<th>instance</th>
<th>FHLMSS</th>
<th>sota</th>
<th>our</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$</td>
<td>V</td>
<td>%$</td>
</tr>
<tr>
<td>soc-firm-hi-tech</td>
<td>63.64</td>
<td>73.63</td>
<td>0.00</td>
</tr>
<tr>
<td>g001207</td>
<td>na</td>
<td>na</td>
<td>0.00</td>
</tr>
<tr>
<td>g000081</td>
<td>na</td>
<td>na</td>
<td>0.00</td>
</tr>
<tr>
<td>ENZYMES_g295</td>
<td>13.82</td>
<td>23.74</td>
<td>0.00</td>
</tr>
<tr>
<td>g000992</td>
<td>79.50</td>
<td>86.13</td>
<td>0.08</td>
</tr>
<tr>
<td>ca-ircs</td>
<td>22.43</td>
<td>22.65</td>
<td>0.00</td>
</tr>
<tr>
<td>bio-disease</td>
<td>6.20</td>
<td>7.91</td>
<td>0.00</td>
</tr>
</tbody>
</table>

The **na** entries indicate the implementation of **FHLMSS** required more RAM than available on the system (128GB) and therefore the data is not available.
Buffered Streaming Edge Partitioning

Adil Chhabra
Heidelberg University, Germany

Marcelo Fonseca Faraj
Heidelberg University, Germany

Christian Schulz
Heidelberg University, Germany

Daniel Seemaier
Karlsruhe Institute of Technology, Germany

Abstract

Addressing the challenges of processing massive graphs, which are prevalent in diverse fields such as social, biological, and technical networks, we introduce HeiStreamE and FreightE, two innovative (buffered) streaming algorithms designed for efficient edge partitioning of large-scale graphs. HeiStreamE utilizes an adapted Split-and-Connect graph model and a Fennel-based multilevel partitioning scheme, while FreightE partitions a hypergraph representation of the input graph. Besides ensuring superior solution quality, these approaches also overcome the limitations of existing algorithms by maintaining linear dependency on the graph size in both time and memory complexity with no dependence on the number of blocks of partition. Our comprehensive experimental analysis demonstrates that HeiStreamE outperforms current streaming algorithms and the re-streaming algorithm 2PS in partitioning quality (replication factor), and is more memory-efficient for real-world networks where the number of edges is far greater than the number of vertices. Further, FreightE is shown to produce fast and efficient partitions, particularly for higher numbers of partition blocks.

2012 ACM Subject Classification Theory of computation → Streaming, sublinear and near linear time algorithms; Theory of computation → Graph algorithms analysis

Keywords and phrases graph partitioning, edge partitioning, streaming, online, buffered partitioning

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.5

1 Introduction

Complex, large graphs, often composed of billions of entities, are employed across multiple fields to model social, biological, navigational, and technical networks. However, processing huge graphs requires extensive computational resources, necessitating the parallel computation of graphs on distributed systems. Large graphs are partitioned into sub-graphs distributed among \( k \) processing elements (PEs). PEs perform computations on a portion of the graph, and communicate with each other through message-passing. Graph partitioning models the distribution of graphs across PEs such that each PE receives approximately the same number of components (vertices) and communication between PEs (via edges between them) is minimized. Edge partitioning, which outperforms traditional vertex partitioning on real-world power-law graphs [22, 32, 34], partitions edges into \( k \) blocks such that vertex replication...
is minimized, hence minimizing the communication needed to synchronize vertex copies. Graph vertex and edge partitioning are NP-hard [8, 21] and there can be no approximation algorithm with a constant ratio factor for general graphs unless P = NP [10]. Thus, heuristic algorithms are used in practice. Further, due to data proliferation, streaming algorithms are increasingly being used to partition huge graphs quickly with low computational resources [1, 3, 17, 18, 19, 24, 25, 35, 50, 51].

Streaming edge partitioning entails the sequential loading of edges for immediate assignment to blocks. One-pass streaming edge partitioners permanently assign edges to blocks during a single sequential pass over the graph’s data stream [41, 53]. Alternatively, buffered streaming algorithms receive and store a buffer of vertices along with their edges before making assignment decisions, thus providing information about future vertices [18, 35], and re-streaming algorithms gather information about the global graph structure [36, 37]. With a few exceptions [37, 53] most streaming edge partitioners have a high time complexity due to a linear dependency on the number of blocks \( k \). However, in recent years, high \( k \) values are frequently used in graph partitioning due to the increasing size of graphs, complexity of computations, and availability of processors. An existing re-streaming edge partitioner, 2PS-L [37], achieves a linear runtime independent of \( k \), but produces lower solution quality than state-of-the-art partitioners and has a linear memory dependence on \( k \). Thus, there remains potential to explore high-quality streaming edge partitioners without a runtime and memory dependency on \( k \).

**Contribution.** We propose HeiStreamE, a buffered streaming algorithm for edge partitioning that leverages the performance efficacy of multilevel algorithms. By employing an adapted version of the SPilt-And-Connect (SPAC) model [33] and solving it with a Fennel-based multilevel scheme [18], our algorithm produces superior solution quality while maintaining time and memory complexities that are linearly dependent on the size of the graph and independent of \( k \). Our results establish the superiority of HeiStreamE over all current streaming algorithms, and even the re-streaming algorithm 2PS [36, 37], in replication factor. These outcomes highlight the considerable potential of our algorithm, positioning it as a promising tool for edge partitioning. We additionally provide an implementation of an efficient streaming edge partitioner, FreightE, which uses streaming hypergraph partitioning [17] to partition edges on the fly. Our experiments demonstrate that FreightE is significantly faster than all competing algorithms, especially for high \( k \) values.

## 2 Preliminaries

### 2.1 Basic Concepts

**Hyper)Graphs.** Let \( G = (V = \{0, \ldots, n - 1\}, E) \) be an undirected graph with no multiple or self-edges, such that \( n = |V|, m = |E| \). Let \( c : V \rightarrow \mathbb{R}_{\geq 0} \) be a vertex-weight function, and let \( \omega : E \rightarrow \mathbb{R}_{\geq 0} \) be an edge-weight function. We generalize \( c \) and \( \omega \) functions to sets, such that \( c(V') = \sum_{v \in V'} c(v) \) and \( \omega(E') = \sum_{e \in E'} \omega(e) \). An edge \( e = (u, v) \) is said to be incident on vertices \( u \) and \( v \). Let \( N(v) = \{u : (v, u) \in E\} \) denote the neighbors of \( v \). A graph \( S = (V', E') \) is said to be a subgraph of \( G = (V, E) \) if \( V' \subseteq V \) and \( E' \subseteq E \cap (V' \times V') \). If \( E' = E \cap (V' \times V') \), \( S \) is an induced subgraph. Let \( d(v) \) be the degree of vertex \( v \) and \( \Delta \) be the maximum degree of \( G \). Let \( H = (V, E) \) be an undirected hypergraph with \( n = |V| \) vertices, \( m = |E| \) hyperedges or nets. A net, unlike an edge of a graph, may consist of more than two vertices, and is defined as a subset of \( V \).
Partitioning. Given a number of blocks \( k \in \mathbb{N}_{\geq 1} \), and an undirected (hyper)graph with positive edge weights, the (hyper)graph partitioning problem pertains to the partitioning of a (hyper)graph into \( k \) smaller (hyper)graphs by assigning the vertices (vertex partitioning) or (hyper)edges (edge partitioning) of the graph to \( k \) mutually exclusive blocks, such that the blocks have roughly the same size and the particular objective function is minimized or maximized. More precisely, a \( k \)-vertex partition of a (hyper)graph partitions \( V \) into \( k \) blocks \( V_1, \ldots, V_k \) such that \( V_1 \cup \cdots \cup V_k = V \) and \( V_i \cap V_j = \emptyset \) for \( i \neq j \). The edge-cut (resp. cut net) of a \( k \)-partition consists of the total weight of the cut edges (resp. cut nets), i.e., (hyper)edges crossing blocks. More formally, let the edge-cut (resp. cut net) be \( \sum_{i<j} \omega(e') \), in which \( E' := \{ e \in E, \exists (u,v) \subseteq e : u \in V_i, v \in V_j, i \neq j \} \) is the cut-set, i.e., the set of all cut edges (resp. cut nets). The balancing constraint demands that the sum of vertex weights in each block does not exceed a threshold associated with some allowed imbalance \( \varepsilon \). More specifically, \( \forall i \in \{1, \ldots, k\} : c(V_i) \leq L_{\max} := \left[(1 + \varepsilon) \frac{c(V)}{n}\right] \). For each net \( e \) of a hypergraph, \( \Lambda(e) := \{ V_i \mid V_i \cap e \neq \emptyset \} \) denotes the connectivity set of \( e \). Further, the connectivity \( \lambda(e) \) of a net \( e \) is the cardinality of its connectivity set, i.e., \( \lambda(e) := |\Lambda(e)| \). The so-called connectivity metric \( (\lambda - 1) \) is computed as \( \sum_{e \in E'} (\lambda(e) - 1) \omega(e) \), where \( E' \) is the cut-set.

Similarly, a \( k \)-edge partition of a graph partitions the edge set \( E \) into \( k \) blocks \( E_1, \ldots, E_k \) such that \( E_1 \cup \cdots \cup E_k = E \) and \( E_i \cap E_j = \emptyset \) for \( i \neq j \). In edge partitioning, a common objective function is the minimization of the replication factor, which is defined as the number of replicated vertices divided by the total number of vertices in the graph. Formally, we define the set \( V(E_i) := \{ v \in V \mid \exists u \in V : (u,v) \in E_i \} \) for each partition \( E_i \) as the number of vertices in \( V \) that have at least one edge incident on them that was assigned to block \( E_i \). Taking the sum of \( |V(E_i)| \) over all \( k \) gives us the total number of vertex replicas generated by the partition. Replication factor is then defined as \( RF(E_1, E_2, \ldots, E_k) = \frac{1}{n} \sum_{i=1}^{k} |V(E_i)| \). Intuitively, a minimized replication factor suggests that vertices are replicated in minimum blocks. Minimum vertex replication, in turn, results in lower synchronization overhead in distributed graph processing due to reduced exchange of vertex state across blocks.

Multilevel Scheme. A successful heuristic for vertex partitioning is the multilevel [14] approach. It recursively computes a clustering and contracts it to coarsen the graph into smaller graphs that maintain the same basic structure as the input graph. An initial partitioning algorithm is applied to the smallest (coarsest) graph and then the contraction is undone. At each level, a local search method is used to improve the partitioning induced by the coarser level. Contracting a cluster of vertices \( C = \{ u_1, \ldots, u_l \} \) involves replacing them with a new vertex \( v \) whose weight is the sum of the weights of the clustered vertices and is connected to all elements \( w \in \bigcup_{i=1}^{l} N(u_i) \), such that \( w \notin C \), with edge weight \( \omega(\{ v, w \}) = \sum_{i=1}^{l} \omega(\{ u_i, w \}) \). This ensures that the transfer of partitions from a coarser to a finer level maintains the edge-cut. The uncontraction of a vertex undoes the contraction. Local search moves vertices between blocks to reduce the objective.

SPAC Transformation. The SPAC transformation [33] provides a means to employ a vertex partitioning tool on a transformed graph \( G' \), which is derived from the original graph \( G \), and subsequently apply the derived vertex partition to establish an edge partition for \( G \). The transformation assumes an undirected, unweighted graph \( G = (V, E) \) as input. The SPAC graph \( G' = (V', E') \) is then constructed in two phases: In the split phase, each vertex \( v \in V \) generates \( d(v) \) split vertices \( S_v := \{ v'_1, \ldots, v'_{d(v)} \} \subseteq V' \). The connect phase introduces two kinds of edges in \( E' \), dominant edges and auxiliary edges. First, it assigns a dominant edge \( e' = (u'_i, v'_j) \) in \( G' \) for each edge \( e = (u, v) \) in \( G \). Dominant edges are created with
infinite weight $\omega(e') = \infty$. Second, it introduces as many auxiliary edges $e''$ as necessary to create a path connecting the vertices in the set $S_v$ for each vertex $v \in V$. Auxiliary edges are created with unitary weight $\omega(e'') = 1$. A visual representation of the SPAC transformation is provided in Figure 2. Due to the infinite weight of dominant edges, vertex partitioning tools usually refrain from splitting them, causing both endpoints of a dominant edge to be grouped in the same block (alternatively, straightforward heuristics can compel both endpoints of dominant edges to belong to the same block). Next, the block assigned to both endpoints of each dominant edge is assigned to the edge in $G$ that induced the corresponding dominant edge, thereby resulting in an edge partition of $G$. The SPAC method is particularly effective in practical scenarios and yields a sound, provable approximation factor under specific balance constraints. Specifically, it approximates the balanced edge partitioning problem within $O(\Delta \sqrt{\log n \log k})$, where $\Delta$ is the maximum degree of $G$ [33].

**Buffered Streaming.** In the buffered streaming model, which is an extended version of the one-pass model, we load a $\delta$-sized buffer or batch of input vertices along with their edges. We make block assignment decisions only after the entire batch has been loaded. In practice, the parameter $\delta$ can be chosen in accordance with memory available on the machine. In our contribution, we use a fixed $\delta$ throughout the algorithm. For a predefined batch size of $\delta$, we load and repeatedly partition $\lceil n/\delta \rceil$ batches.

### 2.2 Related Work

We refer the reader to recent surveys on (hyper)graph partitioning for relevant literature [11, 14, 47]. Here, we focus on the research on streaming vertex and edge partitioning. Most high-quality vertex partitioners for real-world graphs use a multilevel scheme, including KaHIP [44], METIS [28], Scotch [40], and (mt)-KaHyPar [23, 45]. Edge partitioning has been solved directly with multilevel hypergraph partitioners, including PaToH [13], hMETIS [29], KaHyPar [46], Mondriaan [52], MLPart [2], Zoltan [15], SHP [27], UMPa [12], and kPaToH [13].

**Streaming (Hyper)Graph Vertex Partitioning.** Tsourakakis et al. [51] introduce Fennel, a one-pass partitioning heuristic adapted from the clustering objective modularity [9]. Fennel minimizes edge-cuts by placing vertices in partitions with more neighboring vertices. Fennel assigns a vertex $v$ to the block $V_i$ that maximizes the Fennel gain function $|V_i \cap N(v)| - f(|V_i|)$, where $f(|V_i|)$ is a penalty function to respect a balancing threshold. The authors define the Fennel objective with $f(|V_i|) = \alpha \gamma \cdot |V_i|^{\gamma - 1}$, in which $\gamma$ is a free parameter and $\alpha = n^{\frac{k-1}{\gamma-1}}$. After parameter tuning, the authors define $\gamma = \frac{1}{2}$ and $\alpha = \sqrt{\frac{m}{n^{\gamma}}}$. The time complexity of the algorithm depends on $k$ and is given by $O(nk + m)$. Stanton and Kliot [48] propose LDG, a greedy heuristic for streaming vertex partitioning. ReLDG and ReFennel are re-streaming versions of LDG and Fennel [39]. Prioritized re-streaming optimizes the ordering of the streaming process [3]. Furaj and Schulz [18] propose HeiStream, which uses a generalized weighted version of the Fennel gain function in a buffered streaming approach. Eyubov et al. [17] introduce FREIGHT, a streaming hypergraph partitioner that adapts the Fennel objective function to partition vertices of a hypergraph on the fly.

**Streaming Edge Partitioning.** One-pass streaming edge partitioners include hashing-based partitioners like DBH [53], constrained partitioners like Grid and PDS [26], and HDRF, proposed by Petroni et al. [41]. HDRF exploits the skewed degree distribution of real-world graphs by prioritizing vertex replicas of high-degree vertices. HDRF outperforms DBH, Grid and PDS in solution quality, but has a longer runtime. Zhang et al. [54] introduced SNE, a streaming
version of the in-memory edge partitioner NE that utilizes sampling methods. SNE produces better solution quality than HDRF, but with increased memory consumption and runtime [37]. In contrast to one-pass streaming models, RBSEP uses a buffered approach to postpone assignment decisions for edges with limited neighborhood partitioning information available during streaming [49]. Additionally, Mayer et al. [35] introduce ADWISE, a window-based streaming edge partitioner, which uses a dynamic window size that adapts to runtime constraints.

Mayer et al. [36] subsequently propose 2PS-HDRF, a two-phase re-streaming algorithm for edge partitioning, using HDRF as the scoring function in its final partitioning step. The first phase uses a streaming clustering algorithm to gather information about the global graph structure; in the second phase, the graph is re-streamed and partitioned, using information obtained from clustering to make edge partitioning decisions. Mayer et al. [37] modify 2PS-HDRF to propose 2PS-L, which runs in time independent of \( k \). 2PS-L switches from HDRF to a new scoring function in the final partitioning step to remove its dependency on \( k \), and thus achieves a time complexity of \( O(|E|) \). 2PS-L outperforms ADWISE; it is faster than HDRF and 2PS-HDRF, particularly at large \( k \) values, but has lower solution quality. 2PS-HDRF achieves 50% better solution quality than 2PS-L [37].

Sajjad et al. [43] propose HoVerCut, a platform for streaming edge partitioners, which can scale in multi-threaded and distributed systems by decoupling the state from the partitioner. Hoang et al. [24] propose CuSP, a distributed and parallel streaming framework to partition edges based on user-defined policies. CuSP is programmable and can express common streaming edge partitioning strategies from the literature.

3 Buffered Streaming Edge Partitioning

In this section, we present our algorithms, HeiStreamE and FreightE. First, we provide an overview of HeiStreamE’s iterative structure. Subsequently, we detail its input format and buffered graph model, and describe how it uses multilevel vertex partitioning to solve this model. Lastly, we discuss how FreightE builds a hypergraph representation to partition edges using a streaming hypergraph partitioner.

3.1 Overall Algorithm

Our framework draws inspiration from HeiStream [18]. We slide through the input graph \( G \) by iteratively performing the following series of operations until all the edges of \( G \) are assigned to blocks. First, we load a batch composed of \( \delta \) vertices and their associated neighborhood, thereby obtaining a subgraph \( G_b \) contained within the graph \( G \). This operation yields edges connecting vertices within the current batch, and edges connecting vertices in the current batch to vertices streamed in previous batches. Second, we build a model \( \beta_b \) corresponding to \( G_b \), where the edges of \( G_b \) are transformed into vertices. Additionally, we incorporate a representation of block assignments from previous batches into \( \beta_b \). Third, we partition

\begin{algorithm}
1: \textbf{for} \( b \in \{1, \ldots, \lfloor \frac{n}{\delta} \rfloor \} \ \textbf{do}
2: \quad \text{Load subgraph } G_b \text{ from input graph } G
3: \quad \text{Build model } \beta_b \text{ from } G_b
4: \quad \text{Run multilevel vertex partitioning on } \beta_b
5: \quad \text{Permanently assign corresponding edges of } G \text{ in } G_b \text{ to blocks}
\end{algorithm}
Buffered Streaming Edge Partitioning

Figure 1 Detailed structure of HeiStreamE. The algorithm starts by loading a batch graph $G_b$ consisting of vertices and their edges to the current batch and previous batches. Subsequently, it builds a meaningful model $\beta_b$ from the batch graph, transforming edges into vertices, and incorporating a synthetic representation of the assignments made in previous batches. This model is then partitioned using a multilevel algorithm. Lastly, the edges from the loaded batch, which correspond to vertices in the partitioned batch model, are permanently assigned to blocks. This process is repeated for subsequent batches until the entire graph has been partitioned.

3.2 Input and Batch Format
HeiStreamE uses a vertex-centric input format and a buffered streaming approach, which refers to the sequential process of loading and handling the input graph in batches. Within each batch, it loads $\delta$ vertices one at a time along with their neighborhood, where $\delta$ is a parameter that defines the buffer size. Similar input formats are commonly used in streaming algorithms for vertex partitioning [3, 17, 18, 19, 25, 51] and are consistent with graph formats commonly found in publicly available real-world graph datasets, such as the METIS format.

Each batch $b \in \{1, \ldots, \lceil \frac{n}{\delta} \rceil \}$ corresponds to a subgraph denoted as $G_b = (V_b, E_b)$ within the graph $G$. This subgraph is constructed as follows. Its vertex set $V_b$ includes the $\delta$ vertices from the current batch, labeled within the domain $[0, \delta - 1]$, as well as the $p$ vertices from past batches that share at least one edge with vertices in the current batch, labeled within the domain $[\delta, \delta + p - 1]$. Similarly, its edge set $E_b$ comprises of edges with one endpoint in the current batch and the other endpoint in either the current batch or previous batches, expressed as $E_b = \{(u, v) \in E \mid \text{Batch}(u) = b \land \text{Batch}(v) \leq b\}$. Edges with an endpoint in future batches are discarded, ensuring that each edge belongs exclusively to a unique batch graph $G_b$ across all batches.

3.3 Model Construction
Our model construction consists of two steps, detailed in this section. First, using the batch graph $G_b$, we create a corresponding Contracted Split-And-Connect (CSPAC) graph, denoted as $S^*_b$. In $S^*_b$, edges from $G_b$ are directly represented as vertices, while vertices from $G_b$ are indirectly represented as edges. Subsequently, we create a graph model $\beta_b$ based on $S^*_b$, which incorporates a representation of block assignments from prior batches.
Contract Dominant Edges
Split-and-Connect
Input Graph $G$
SPA C Graph $G'$
CSPAC Graph $G^*$

Figure 2 Building SPAC and CSPAC Graphs: The SPAC graph, denoted as $G'$, features $d(v)$ split vertices for every vertex $v$ in the original graph $G$. These split vertices are represented in the same color as the vertex they originate from. Every (thick green) edge from $G$ is directly converted into a distinct dominant (thick green) edge in $G'$ that connects corresponding split vertices. The auxiliary (thin) edges in $G'$, which create a path between split vertices, are depicted in the same color as the split vertices they link. The CSPAC graph $G^*$ is formed by contracting the dominant edges of $G'$. The vertices in $G^*$ represent the edges in $G$, while the edges in $G^*$ mirror the auxiliary edges in $G'$.

CSPAC Transformation. The CSPAC transformation is a faster and more condensed variant of the SPAC transformation conceived by Li et al. [33], as described in Section 2.1. Below we explain how the SPAC transformation evolves into the CSPAC transformation, and how this transformation is applied to $G_b$ to yield the CSPAC graph $S^*_b$.

The SPAC graph $G'$ of a graph $G$ consists of dominant edges, i.e., edges with a weight of infinity that have a one-to-one correspondence with edges of the original graph, and auxiliary edges, which define a path between vertices of the SPAC graph. The CSPAC transformation is derived from the SPAC graph by contracting the dominant edges into vertices. Due to construction, the dominant edges in $G'$ do not share any endpoints; they effectively form a matching, ensuring a consistent contraction. Further, each endpoint of the auxiliary edges is incident to a single dominant edge. Thus, the contraction of all dominant edges produces a coarser graph, in which every vertex represents a unique edge in the original graph $G$, and their connections correspond to the auxiliary edges in graph $G'$. The CSPAC transformation is illustrated in Figure 2.

In HeiStreamE, we derive the CSPAC graph $S^*_b$ directly from the batch graph $G_b$, bypassing the construction of the intermediary SPAC graph. The procedure for this direct transformation is as follows. For each vertex $u \in V_b$, each of its outgoing edges $e = (u, v) \in E_b$ induces a vertex $u^*$ in $S^*_b$, if $u < v$ to avoid redundancy. As each vertex is constructed, it is connected to at most two other vertices in $S^*_b$ that are induced by other outgoing edges of $u$ in $G_b$ to form a path. This way, each undirected edge $e \in E_b$, which would have induced a dominant edge in the SPAC graph, is represented as a vertex in $S^*_b$. Further, each auxiliary edge of the SPAC graph is directly integrated into $S^*_b$ to form paths between vertices in $S^*_b$.

The direct construction of the CSPAC graph maintains the same computational complexity as the SPAC construction alone, specifically $O(|V_b| + |E_b|)$, where $|V_b|$ and $|E_b|$ are the number of vertices and edges of $G_b$ respectively. In practice, building $S^*_b$ directly from $G_b$ offers a time-saving advantage compared to the alternative method of building the SPAC graph and then contracting the dominant edges.

In each batch, $S^*_b$ has $|E_b|$ vertices and $2|E_b| - |V_b|$ edges, i.e., it is linear in the size of the batch graph $G_b$. In contrast to the SPAC graph, the CSPAC graph has fewer vertices and edges. Further, where a vertex partitioner might cut a dominant edge of the SPAC graph, every vertex partition of the CSPAC graph corresponds to a valid edge partition of the original graph. Thus, there is no need for a verification step to transform a vertex partition of $S^*_b$ into an edge partition of $G_b$. 
Theorem 1 shows that when computing a vertex partition of $S^*_b$ to minimize the edge-cut, the corresponding edge partition of $G_b$ will also have a minimized number of vertex replicas. The SPAC approximation factor shown by Li et al. [33] is also directly valid for $S^*_b$.

**Theorem 1.** For any vertex partition $vp(S^*_b)$ of the CSPAC graph $S^*_b$ with edge-cut $cost(vp(S^*_b))$, there exists a corresponding edge partition $ep(G_b)$ of the batch graph $G_b$ with a number of vertex replicas $cost(ep(G_b))$, satisfying $cost(ep(G_b)) \leq cost(vp(S^*_b))$, which establishes a lower bound spanning the set of possible CSPAC graphs $S^*_b$ associated with $G_b$.

*Proof.* See Appendix B.

**Integrating Connectivity Information.** Directly partitioning the CSPAC graph $S^*_b$ limits the partitioner’s view to the current batch, as $S^*_b$ does not take into account block assignments from previous batches. Specifically, when assigning a block to a vertex $u^* \in S^*_b$ induced by an edge $e = (u, v) \in E_b$, where $u$ is a vertex of the current batch and $v$ is a vertex of some previous batch, the partitioner might replicate $v$ into a new block in the absence of global information. To solve this, we extend the CSPAC graph $S^*_b$ with connectivity information derived from previous batch assignments to obtain the graph model $\beta_b$. We construct $\beta_b$ by augmenting $S^*_b$ with $k$ artificial vertices representing the $k$ partition blocks.

Each artificial vertex $i$ corresponds to an existing partition block $E_i$ in its current state, i.e., the weight of each artificial vertex $i$ is the weight of its corresponding block $E_i$ filled with edges that have been assigned to it from previous batches. An edge $e = (u, v) \in E_b$ of the current batch is represented in our model $\beta_b$ as a vertex that connects to an artificial vertex $i$ if $v$ is a vertex of a previous batch and an edge incident to $v$ has already been assigned to block $E_i$. However, in a streaming setting, there is limited knowledge of edge connectivity, i.e., it is not possible to directly determine which edges from previous batches are adjacent to edges from the current batch. To overcome this constraint, we maintain an array $B$ of size $n$ throughout the streaming process. This array records, for each vertex $u \in V$, block(s) assigned to edges incident on it. Then, in $\beta_b$, a vertex $u^* \in S^*_b$ induced by an edge $e = (u, v) \in E_b$ is connected to any artificial vertices $i$ representing blocks recorded in $B[u]$. An added benefit of this model construction is that we do not need to maintain a data structure of size $m$ in our algorithm.
We propose three configurations for \( \beta_b \), which vary in how they use \( B \), that we name, in decreasing level of exactness, maximal, \( r \)-Subset, and minimal. Each configuration has a runtime, memory and solution-quality trade-off. Let \( u^* \) be a vertex in \( S_b \) induced by an edge \( e = (u, v) \in E_b \). In the maximal model \( \beta_b \), \( u^* \) is connected to all artificial vertices \( i \) representing blocks recorded in \( B[v] \). The memory required to store the array \( B \) in the maximal setup is \( O(nk) \). Theorem 2 demonstrates that computing a vertex partition of the maximal model \( \beta_b \) to minimize edge-cut corresponds to an edge partition of \( G_b \) that contributes a minimized number of new vertex replicas to the overall edge partition of the input graph \( G \). In the \( r \)-Subset model, \( u^* \) is connected to a sample of \( r \) artificial vertices \( i \) representing blocks recorded in \( B[v] \), where \( r \) is a parameter. Here, \( B \) is identical as in the maximal setup, but the model \( \beta_b \) is more concise, allowing for a faster partitioning phase. In the minimal model, \( u^* \) is connected to a single artificial vertex \( i \) representing the most recent block assigned to \( v \) in a previous batch. We only store the latest assignment per vertex in \( B \), thus the memory requirement is \( O(n) \), and the model \( \beta_b \) is also more concise than in the other two setups. We illustrate the various configurations in Figure 3.

\[ \text{Theorem 2. For any vertex partition } vp(\beta_b) \text{ of the maximal model } \beta_b \text{ with edge-cut cost}(vp(\beta_b)), \text{ there exists a corresponding edge partition } ep(G_b) \text{ of the batch graph } G_b \text{ that, when incorporated into the already partitioned section of the input graph } G, \text{ introduces a number } \gamma \text{ of new vertex replicas, satisfying } \gamma \leq \text{cost}(vp(\beta_b)). \]

Proof. See Appendix B.

### 3.4 Partitioning

In this section, we describe how we partition our model \( \beta_b \). We employ a vertex partitioning algorithm on \( \beta_b \), specifically an adapted version of the multilevel weighted Fennel algorithm utilized in HeiStream [18]. We describe this algorithm and then present a modification to the initial partitioning phase to enhance our runtime performance. Lastly, we discuss possible adaptations to the Fennel parameter \( \alpha \).

**Multilevel Fennel.** Each per-batch graph model \( \beta_b \) is partitioned using a multilevel partitioning scheme consisting of three successive phases, coarsening, initial partitioning, and uncoarsening.

In the coarsening phase, the algorithm computes a clustering and contracts the graph at each level until it is smaller than a specified threshold. These clusters are computed with label propagation while adhering to size constraints [38]. The clustering algorithm ignores artificial vertices and edges during the coarsening phase, to ensure that they are never contracted and that previous block assignment decisions are available at the coarsest level. For a graph with \( n \) vertices and \( m \) edges, a single round of size-constrained label propagation can be executed in \( O(n + m) \) time. Initially, each vertex is placed in its own cluster, and in subsequent rounds vertices move to the cluster with the strongest connection, with a maximum of \( L \) rounds, where \( L \) is a tuning parameter. The coarsening phase ends as soon as the graph has fewer vertices than a threshold of \( O(\max(\lfloor |\beta_b|/k\rfloor, k)) \), where \( |\beta_b| \) is the number of vertices of \( \beta_b \). For large buffer sizes this threshold simplifies to \( O(\lfloor |\beta_b|/k\rfloor) \), while for small buffer sizes, it becomes \( O(k) \).

Following the coarsening phase, an initial partitioning assigns all non-artificial vertices to blocks using the generalized Fennel algorithm [18] with a strict balancing constraint \( L_{\max} \). After initial partitioning, the current solution is transferred to the next finer level by mapping the block assignment of each coarse vertex to its constituent vertices at the
Buffered Streaming Edge Partitioning

Subsequently, a local search algorithm is applied at each level, which utilizes the size-constrained label propagation approach employed in the contraction phase but with a modified objective function. Specifically, when visiting a non-artificial vertex, we reassign it to a neighboring block to maximize the generalized Fennel gain function while strictly adhering to the balancing constraint \( L_{\text{max}} \), considering only adjacent blocks in contrast to the initial partitioning which considers all blocks. This ensures that a single algorithm round remains linear in the current level’s size. Artificial vertices remain stationary during the process, but unlike during coarsening, they are not excluded from label propagation, as they contribute to the generalized Fennel gain function.

**Faster Initial Partitioning.** We adopt a modified Fennel function to enhance the initial partitioning step in HeiStream\textsuperscript{E} and demonstrate that this approach produces a better runtime without a decrease in solution quality.

For small buffer sizes, HeiStream\textsuperscript{E} has a linear dependency on \( k \) for overall partitioning time, as during initial partitioning each vertex \( u \in \beta_b \) is assigned to the block with the highest score among all \( k \) blocks. To address this dependency, we adopt Eyubov et al.’s strategy \[17\] from streaming hypergraph partitioning to evaluate scores more efficiently, removing the runtime dependency on \( k \). For the current vertex \( u \in \beta_b \), we categorize \( k \) blocks \( V_i \) into two sets, \( S_1 \) and \( S_2 \), where \( V_i \in S_1 \) if a neighbor of \( u \) was assigned to it, and \( V_i \in S_2 \) otherwise. This allows us to determine the blocks \( V_{\text{max}} \) and \( V'_{\text{max}} \) that respectively maximize Equation (1) and Equation (2). Then, the block that maximizes the generalized Fennel function is \( \max(V_{\text{max}}, V'_{\text{max}}) \) where

\[
V_{\text{max}} = \arg\max_{i \in S_1} \left\{ \sum_{v \in V_i \cap N(u)} \omega(u, v) - c(u) \alpha f(c(V_i)) \right\} \tag{1}
\]

\[
V'_{\text{max}} = \arg\max_{i \in S_2} \left\{ -c(u) \gamma f(c(V_i)) \right\} \tag{2}
\]

As \( c(u) \) remains constant, determining the block \( V_i \in S_2 \) that maximizes Equation (2) is equivalent to finding the block \( V_i \) minimizing \( f(c(V_i)) = \alpha * \gamma * c(V_i)^{-1} \), specifically, the \( V_i \) with the lowest block weight \( c(V_i) \). In our scheme, \( c(V_i) = c(E_i) \), that is, the number of edges assigned to block \( E_i \) of the overall edge partition. This optimized process, facilitated by a priority queue, reduces the evaluation of blocks for each vertex to those assigned to its neighbors and the minimum weight block overall. It results in an optimal block \( V_i \) for maximizing the generalized Fennel gain function in \( O(d(u) + \log(k)) \) time using a binary heap priority queue or \( O(d(u)) \) with a bucket priority queue, as suggested in \[17\], ultimately yielding an overall linear time complexity. With this enhanced approach, the runtime becomes independent of the parameter \( k \).

**The Parameter \( \alpha \).** The authors of Fennel \[51\], define the parameter \( \alpha = \sqrt{k} \frac{m}{n^{3/2}} \) for vertex partitioning of an input graph \( G = (V, E) \) with \( n \) vertices and \( m \) edges. However this choice of \( \alpha \) is not directly applicable for edge partitioning in our CSPAC-based model. If we built a single model \( \beta_b \) for the whole graph \( G \) at once, it would have \( n^* = m \) vertices and a number \( m^* \) of edges equal to the number of auxiliary edges in \( \beta_b \). While we know \( n^* \) immediately, we cannot directly obtain \( m^* \) without visiting all vertices of the whole graph. Thus, we need to estimate \( m^* \).
We thus have three distinct approaches for determining the Fennel parameter $\alpha$. In the static $\alpha$ method, we keep $\alpha$ constant across all batches, setting it to $\alpha = \sqrt{\frac{n^*}{m_{\text{approx}}}}$, where $n^* = m/2$, $m_{\text{approx}} = y + n^*$, and $y$ is a tuning parameter. For the batch $\alpha$ approach, we update $\alpha$ for each batch, calculating it as $\alpha = \sqrt{\frac{m_s}{n_s}}$, where $n_s$ and $m_s$ are the number of vertices and edges of each CSPAC graph $S_b^*$ respectively. The dynamic $\alpha$ method also updates $\alpha$ per batch. It begins with the static $\alpha$ setting and refines $m_{\text{approx}}$ in each batch by computing the number of auxiliary edges, determined through counting vertices of degree less than or equal to 2.

3.5 FreightE

In addition to HeiStreamE, we present FreightE, a fast streaming edge partitioner that uses streaming hypergraph partitioning to assign blocks to edges on the fly. In general, a hypergraph vertex partitioner can partition edges of an input graph $G$ by first transforming it into its dual hypergraph representation $H$, where each edge of $G$ is a hypervertex, and each vertex of $G$ induces a hyperedge spanning its incident edges. Then, a hypergraph vertex partitioner that assigns the hypervertices of $H$ into $k$ blocks, while optimizing for the connectivity metric, directly provides an edge partition of $G$. The intuition behind this approach is that a hypergraph vertex partitioner that optimizes for the connectivity metric directly optimizes the replication factor of the underlying edge partition [14]. In FreightE, we perform the transformation of $G$ into $H$ on the fly as follows. At a given step, we read a vertex of the input graph (METIS format) along with its neighborhood. Each undirected edge in the neighborhood is treated as a unique hypervertex of the hypergraph, and permanently assigned to a block using the FREIGHT streaming hypergraph partitioner [17]. This process is repeated until all vertices along with their neighborhoods are visited, at which point each edge of the input graph is assigned to a block. Following the FREIGHT partitioner, the overall runtime of FreightE is $O(n + m)$, and memory complexity is $O(m + k)$. In comparison to HeiStreamE, FreightE is faster, as it does not require the construction of an equivalent CSPAC graph, and requires less memory unless $m \gg n$.

4 Experimental Evaluation

Setup. We implemented HeiStreamE and FreightE inside the KaHIP framework (using C++) and compiled it using gcc 9.3 with full optimization enabled (-O3 flag). Except for the largest graph instance gsh-2015, all experiments were performed on a single core of a machine consisting of a sixteen-core Intel Xeon Silver 4216 processor running at 2.1 GHz, 100 GB of main memory, 16 MB of L2-Cache, and 22 MB of L3-Cache running Ubuntu 20.04.1. To facilitate algorithms that required greater memory, gsh-2015 was run on a single core of an alternate machine, consisting of a 64-core AMD EPYC 7702P Processor containing 1 TB of main memory.

Baselines. We compare HeiStreamE and FreightE against the state-of-the-art streaming algorithm HDRF, as well as the re-streaming algorithms 2PS-HDRF [36] and 2PS-L [37], which require three passes over the input graph. We exclude the following algorithms: SNE [54], as it fails to execute for $k > 127$ and is outperformed by 2PS-HDRF [36]; DBH [53], as it ignores past partition assignments and thus has poor solution quality; ADWISE and RBSEP as they have limited global information during streaming, and 2PS-L outperforms them [37].

For comparison with competitors, we obtained implementations of 2PS-HDRF and 2PS-L from their official repository, which also provides an implementation of HDRF. We configure all competitor algorithms with the optimal settings provided by the authors. 2PS-HDRF,
**Buffered Streaming Edge Partitioning**

2PS-L and HDRF require a vertex-to-partition table of size $O(n \times k)$, which stores the blocks that each vertex was replicated on. To optimize memory usage, these partitioners can be built with the number of partitions at compile time to only allocate the required amount of memory. Thus, we re-compile them with the CMake flag for the number of blocks for each $k$ value in our experiments. We set the HDRF scoring function parameter $\lambda = 1.1$, similar to the authors of 2PS [36]. The provided codes for 2PS-HDRF, 2PS-L, and HDRF in their official repositories set a hard-coded soft limit on the number of partitions to 256, which we override to test the algorithms with larger $k$ values.

All competitors read a binary edge list format with 32-bit vertex IDs. This allows for faster IO during program execution. Additionally, all competitor programs offer a converter which can load a graph in the standard edge list format, convert it into the binary edge list format, and write it to memory before proceeding. For a fair comparison with HeiStreamE and FreightE, which are capable of reading both text-based (METIS) and binary (adjacency) graphs, we perform experiments with binary graph formats only. Further, we exclude the IO time since the objective of the experiments is to measure the performance of the partitioners and not IO efficiency. Similarly, we exclude the time it takes to convert the graphs into the binary format for all partitioners.

**Instances.** Our graph instances for experiments are shown in Appendix Table 2 and are sourced from Ref. [4, 5, 6, 7, 20, 31, 42]. All instances evaluated have been used for benchmarking in previous works on graph partitioning. From these graph instances, we construct three disjoint sets: a tuning set for parameter study experiments, a test set for comparison against state-of-the-art and a set of huge graphs, for which in-memory partitioners ran out of memory on our machine. We set the number of blocks to $k = \{2^1, 2^2, \ldots, 2^{14}\}$ for all experiments except those on huge graphs, for which $k$ values are shown in Table 1. We allow an imbalance of $\varepsilon = 3\%$ for all partitioners. While streaming, we use the natural order of the vertices in these graphs.

**Methodology.** We measure running time, replication factor and memory consumption, i.e., the maximum resident set size for the executed process. When averaging over all instances, we use the geometric mean to give every instance the same influence on the final score. Further, we average all results of each algorithm grouped by $k$, to explore performance with increasing $k$ values. Let the runtime, replication factor or memory consumption be denoted by the score $\sigma_A$ for some $k$-partition generated by an algorithm $A$. We express this score relative to others using the following tools: improvement over an algorithm $B$, computed as a percentage $\left(\frac{\sigma_A}{\sigma_B} - 1\right) \times 100\%$ and relative value over an algorithm $B$, computed as $\frac{\sigma_A}{\sigma_B}$. Additionally, we present performance profiles by Dolan and Moré [16] to benchmark our algorithms. These profiles relate the running time (resp. solution quality, memory) of the slower (resp. worse) algorithms to the fastest (resp. best) one on a per-instance basis, rather than grouped by $k$. Their $x$-axis shows a factor $\tau$ while their $y$-axis shows the percentage of instances for which an algorithm has up to $\tau$ times the running time (resp. solution quality, memory) of the fastest (resp. best) algorithm.

**Parameter Tuning.** We tuned the parameters used by HeiStreamE, namely the selection of the batch graph model $\beta_b$ mode and the choice of Fennel $\alpha$, through experiments run on the Tuning Set (see Appendix Table 2). Due to space constraints we only give a short summary. Our results support the use of the minimal mode for $\beta_b$ and batch $\alpha$ for Fennel’s $\alpha$ parameter for subsequent experiments. Other parameters for the multilevel partitioning
Figure 4 Comparison of HeiStreamE and FreightE with 2PS-HDRF, 2PS-L and HDRF on the Test Set in Appendix Table 2 using performance profiles. Let \(\mathcal{A}\) be the set of all algorithms, \(\mathcal{I}\) the set of instances, and \(q_A(I)\) the quality of algorithm \(A \in \mathcal{A}\) on instance \(I \in \mathcal{I}\). For each algorithm \(A\), we plot the fraction of instances \(|\mathcal{I}_A(\tau)|/|\mathcal{I}|\) (y-axis) where \(\mathcal{I}_A(\tau) := \{I \in \mathcal{I} | q_A \leq \tau \cdot \min_{A' \in \mathcal{A}} q_{A'}(I)\}\) and \(\tau\) is on the x-axis. Includes all \(k\) values. Note the logarithmic scale in the final third of the plots. Memory consumption is measured as the maximum resident set size of the program execution. 2PS-HDRF and 2PS-L have the same memory consumption.

scheme align with optimal values from HeiStream [18]. The minimal model mode produces a solution quality improvement of 11.73% over the baseline of using no graph model, which is comparable to the 12.8% increase achieved by the maximal mode, while being 1.41× and 1.29× faster than the maximal and r-Subset mode respectively. On average, across all instances and all \(k\) values, batch \(\alpha\) produces 0.86% and 3.27% better solution quality than static and dynamic \(\alpha\) respectively. For \(k > 32\), these averages increase to 2.1% and 5.2%, respectively.

Comparison with State-of-the-Art

We now provide experiments in which we compare HeiStreamE and FreightE against the current state-of-the-art algorithms for (re)streaming edge partitioning, namely, HDRF, 2PS-HDRF and 2PS-L. These experiments were performed on the Test Set and the Huge Set of graphs in Appendix Table 2. Figure 4 gives performance profiles for the Test Set and Table 1 gives detailed per instance results for instances of the Huge Set. We distinguish between buffer sizes by defining, for example, HeiStreamE(32x) as HeiStreamE with a buffer size of 32 \(768 = 32 \times x\) vertices, where \(x = 1024\).

Replication Factor. HeiStreamE(32x) produces better solution quality than state-of-the-art (re)streaming edge partitioners for all \(k\) values. The results from our Test Set demonstrate that HeiStreamE(32x) achieves an average improvement in solution quality of 7.56% (or 13.69% when using HeiStreamE(256x)) compared to 2PS-HDRF, which produces the next best solution quality. As displayed in Figure 4(a), HeiStreamE(32x) produces better solution quality than 2PS-HDRF in approximately 80% of all Test Set instances. The largest improvement in replication factor that we observed for HeiStreamE(32x) over 2PS-HDRF is of \(\approx 53\%\) on graph circuit5M for \(k \geq 8192\). Further, HeiStreamE(32x) achieves an average improvement in solution quality of 51.84% percent over 2PS-L, and an average improvement of 202.86% over HDRF, the only other on-the-fly streaming algorithm presented here. These results from the Test Set are reflected in our experiments on huge graphs, shown in Table 1: HeiStreamE(32x) and HeiStreamE(256x) produce the best solution quality in most instances, providing the best solution quality for all \(k\) values for five of the six huge graphs. While FreightE produces
Figure 5 Comparison of HeiStreamE and FreightE with 2PS-HDRF, 2PS-L and HDRF on the Test Set in Appendix Table 2 using performance profiles (see Figure 4 caption for detailed description). Only includes runs with $k \geq 512$. Note the logarithmic scale in the final third of the plots. Memory consumption is measured as the maximum resident set size of the program execution. 2PS-HDRF and 2PS-L have the same memory consumption.

lower solution quality than 2PS-L and 2PS-HDRF on most Test Set instances, for $k \geq 512$, FreightE produces 9.88% higher solution quality than 2PS-L on average. FreightE produces 87.28% better solution quality than HDRF on average across all Test Set instances and $k$ values. The same trends for FreightE are reflected in our results on huge graphs.

**Runtime.** HeiStreamE(32x) is on average slower than 2PS-HDRF and HDRF for $k \leq 256$; however, since its runtime is not linearly dependent on $k$, HeiStreamE is substantially faster than 2PS-HDRF and HDRF for higher $k$ values (Figure 5b). On the Test Set, HeiStreamE(32x) is on average 6.7× faster than 2PS-HDRF and 8.7× faster than HDRF for $k \geq 512$. Compared to 2PS-L, whose time complexity is also independent of $k$, HeiStreamE(32x) is on average slower across all instances. Similarly, in our experiments on huge graphs, HeiStreamE(32x) and HeiStreamE(256x) are faster than 2PS-HDRF and HDRF for high $k$ values, but slower than 2PS-L. On the Test Set, FreightE is the fastest algorithm among competitors; it is 1.3× faster than 2PS-L which is the next fastest algorithm. Additionally, it is 9.9× faster than HDRF and 8× faster than 2PS-L on average across all Test Set instances. In our experiments on huge graphs, FreightE is faster than HDRF and 2PS-HDRF on average across all instances, and the fastest algorithm for $k \geq 2048$.

**Memory Consumption.** Since HeiStreamE(32x) uses a buffered streaming approach, it consumes on average more memory than 2PS-HDRF, 2PS-L and HDRF for $k \leq 256$ on the Test Set; however, since its memory consumption is not asymptotically dependent on $m$ or $k$, HeiStreamE(32x) consumes significantly less memory than 2PS-HDRF, 2PS-L, and HDRF for higher $k$ values. On average, for $k \geq 512$, 2PS-HDRF and 2PS-L use 3.0× more memory than HeiStreamE(32x), and HDRF uses 2.7× more memory than HeiStreamE(32x). Notably, while HeiStreamE(32x) consumes 1.46× more memory on average than 2PS-HDRF and 2PS-L for $k \leq 256$ on the Test Set, in three out of six of the huge graphs, it is more memory efficient than them across all $k$ values. On the Test Set, FreightE consumes significantly less memory than all competitors as shown in Figure 4c. It consumes 4.9× less memory than HDRF and 7.2× less memory than 2PS-L and 2PS-HDRF on average across all instances and all $k$ values. Further, on average, FreightE consumes 16× less memory than 2PS-L and 2PS-HDRF for $k \geq 512$ (Figure 5c). Since FreightE’s memory consumption is linearly dependent on $m$, it is less memory efficient on the huge graphs, which, like other real-world graphs, tend to have many more edges than vertices. FreightE uses more memory than competitors for $k \leq 256$ in five out of six of the huge graphs, but is more memory efficient at
high $k$ due to its memory being asymptotically independent of $k$. In our experiments on huge graphs, 2PS-HDRF, 2PS-L and HDRF exceed the available memory on the machine at $k \geq 1096$ for uk-2007-05 and webbase-2001, at $k \geq 8192$ for com-friendster. At $k \geq 16384$ they exceed the available memory for all graphs, while HeiStreamE and Freigeht's memory consumption is predictable and consistent across $k$.

Table 1: Results of experiments on the Huge Set in Appendix Table 2. Here, we compare HeiStreamE(32x), i.e., HeiStreamE with a buffer size of 32 · 1024 (resp. HeiStreamE(256x)), and Freigeht with state-of-the-art (re)streaming edge partitioners on huge graph instances, displaying the Replication Factor $RF$, Running Time $RT$ [s] and Memory Consumption $Mem$ [GB]. The best solution for each instance is marked with an asterisk.

<table>
<thead>
<tr>
<th>$G$</th>
<th>$k$</th>
<th>HeiStreamE(32x)</th>
<th>HeiStreamE(256x)</th>
<th>Freigeht</th>
<th>2PS-HDRF</th>
<th>2PS-L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>128</td>
<td>2.35</td>
<td>1.90</td>
<td>1.91</td>
<td>1.27</td>
<td>1.27</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>1.11</td>
<td>1.09</td>
<td>1.09</td>
<td>1.09</td>
<td>1.09</td>
</tr>
<tr>
<td>4096</td>
<td>2048</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
</tr>
<tr>
<td>16384</td>
<td>32768</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
<td>1.11</td>
</tr>
</tbody>
</table>

A. Chhabra, M. Fonseca Faraj, C. Schulz, and D. Seemaier
5 Conclusion

In this work, we propose **HeiStreamE**, a buffered streaming edge partitioner that achieves state-of-the-art solution quality, and **FreightE**, a highly efficient streaming edge partitioner that uses streaming hypergraph partitioning to assign blocks to edges on the fly. **HeiStreamE** processes the input graph in batches, constructs a novel graph transformation on the per-batch graph model, extends it with global partitioning information, and partitions it with a multilevel scheme. Aside from hashing-based streaming partitioners, which have poor solution quality, **HeiStreamE** and **FreightE** are the only known streaming edge partitioners whose runtime and memory consumption are both linear and asymptotically independent of the number of blocks of partition \( k \). Our experiments demonstrate that **HeiStreamE** and **FreightE** consistently outperform all existing (re)streaming edge partitioners with regard to vertex replication metrics and runtime respectively. Additionally, **HeiStreamE** uses less memory than high-quality (re)streaming edge partitioners at \( k \geq 256 \) or when the graph has far more edges than vertices, as is the case for most real-world networks. Our findings underscore **HeiStreamE** as a highly memory-efficient and effective solution for streaming edge partitioning of large-scale graphs.

References


Buffered Streaming Edge Partitioning


Table 2 Graphs used for experiments. The roadNet graphs, wiki graphs, web-Google, web-NotreDame, and all social, co-purchasing, and autonomous systems graphs were obtained from the publicly available SNAP dataset [31]. We also used graphs from the 10th DIMACS Implementation Challenge, namely eu-2005, in-2004 and uk-2007-05 [4]. Any remaining graphs are available on the network repository website [42] or on the Laboratory for Web Algorithms website [5, 6, 7]. For our experiments, we converted these graphs to a vertex-stream format (METIS) while removing parallel edges, self-loops, and directions, and assigning unitary weight to all vertices and edges. If a graph was unavailable in an equivalent edge list format, we converted the METIS format into the edge list format while preserving edge order for fairness during streaming.

\[
\begin{array}{llll}
\text{Graph} & n & m & \text{Type} \\
\hline
\text{bicircuit} & 105,670 & 203,734 & \text{Circuit} \\
\text{coAuthorsCiteseer} & 227,320 & 814,134 & \text{Citations} \\
\text{coAuthorsDBLP} & 299,067 & 977,676 & \text{Citations} \\
\text{com-DBLP} & 317,080 & 1,049,866 & \text{Social} \\
\text{roadNet-PA} & 1,088,092 & 1,541,898 & \text{Roads} \\
\text{web-_google} & 356,648 & 2,093,324 & \text{Web} \\
\text{amazon0601} & 403,394 & 2,443,498 & \text{Co-Purch.} \\
\text{com-Youtube} & 1,134,890 & 2,987,624 & \text{Social} \\
\text{Amazon-2008} & 735,323 & 3,523,472 & \text{Similarity} \\
\text{soc-lastfm} & 1,191,805 & 4,519,330 & \text{Social} \\
\text{as-Skitter} & 554,930 & 5,797,633 & \text{Aut. Syst.} \\
\text{italy-osm} & 6,686,493 & 7,013,978 & \text{Roads} \\
\text{in-2004} & 1,382,908 & 13,591,473 & \text{Web} \\
\text{ML_Laplace} & 377,002 & 13,656,485 & \text{Mesh} \\
\text{webbase2001} & 118,142,155 & 854,809,761 & \text{Web} \\
\text{ti-2004} & 41,291,594 & 1,027,474,947 & \text{Web} \\
\text{com-friendster} & 65,698,366 & 1,806,067,135 & \text{Social} \\
\text{uk-2005} & 50,636,154 & 1,810,063,330 & \text{Web} \\
\text{uk-2007-05} & 105,406,555 & 3,301,876,564 & \text{Web} \\
\text{gsh-2015} & 988,490,691 & 25,690,705,118 & \text{Web} \\
\end{array}
\]

\[
\begin{array}{llll}
\text{Graph} & n & m & \text{Type} \\
\hline
\text{Dubcoval} & 16,129 & 118,440 & \text{Mesh} \\
\text{DBLP-2010} & 300,647 & 807,700 & \text{Citations} \\
\text{com-Amazon} & 334,863 & 925,872 & \text{Social} \\
\text{web-NotreDame} & 325,729 & 1,090,188 & \text{Web} \\
\text{citationCiteseer} & 268,495 & 1,156,647 & \text{Citations} \\
\text{wiki-Talk} & 232,314 & 1,458,806 & \text{Web} \\
\text{roadNet-TX} & 1,379,917 & 1,921,660 & \text{Roads} \\
\text{amazon0312} & 400,727 & 2,349,869 & \text{Co-Purch.} \\
\text{amazon0505} & 410,236 & 2,439,437 & \text{Co-Purch.} \\
\text{roadNet-CA} & 1,965,206 & 2,766,607 & \text{Roads} \\
\text{G3_circuit} & 1,585,478 & 3,037,674 & \text{Circuit} \\
\text{soc-Eijcek} & 2,523,386 & 7,918,810 & \text{Social} \\
\text{great-britain-osm} & 7,738,822 & 8,156,517 & \text{Roads} \\
\text{FullChip} & 2,986,999 & 11,817,567 & \text{Circuit} \\
\text{coPapersDBLP} & 540,486 & 15,245,729 & \text{Citations} \\
\text{coPapersCiteseer} & 434,102 & 16,036,720 & \text{Citations} \\
\text{eu-2005} & 862,664 & 16,138,468 & \text{Web} \\
\text{soc-psoc} & 1,632,803 & 22,301,964 & \text{Social} \\
\text{wikis-topcats} & 1,791,489 & 25,444,207 & \text{Social} \\
\text{circuit5M} & 5,558,311 & 26,983,926 & \text{Circuit} \\
\text{com-LJ} & 3,997,962 & 34,681,189 & \text{Social} \\
\text{soc-LiveJournal} & 4,846,609 & 42,851,237 & \text{Social} \\
\text{Ljournal-2008} & 5,363,186 & 49,514,271 & \text{Social} \\
\text{ca-hollywood2009} & 1,069,126 & 56,306,653 & \text{Roads} \\
\text{Flan1565} & 1,564,794 & 57,920,625 & \text{Mesh} \\
\text{Bump2011} & 2,852,430 & 62,499,240 & \text{Mesh} \\
\text{com-Orkut} & 3,072,441 & 117,185,083 & \text{Social} \\
\text{HV15R} & 2,017,169 & 162,357,569 & \text{Mesh} \\
\end{array}
\]

\section{Omitted Proofs}

\textbf{Theorem.} For any vertex partition \( vp(S_b^*) \) of the CSPAC graph \( S_b^* \) with edge-cut cost \( \text{cost}(vp(S_b^*)) \), there exists a corresponding edge partition \( ep(G_b) \) of the batch graph \( G_b \) with a number of vertex replicas \( \text{cost}(ep(G_b)) \), satisfying \( \text{cost}(ep(G_b)) \leq \text{cost}(vp(S_b^*)) \), which establishes a lower bound spanning the set of possible CSPAC graphs \( S_b^* \) associated with \( G_b \).
Proof. Our proof can be delineated through three consecutive claims. (i) The existence of a singular edge partition \( ep(G_b) \) that corresponds to a specified vertex partition \( vp(S_b^*) \). (ii) The general validity of the inequality \( cost(ep(G_b)) \leq cost(vp(S_b^*)) \). (iii) The validity of the equality \( cost(ep(G_b)) = cost(vp(S_b^*)) \) for a specific CSPAC graph \( S_b^* \).

Claim (i) trivially holds, as \( ep(G_b) \) stems directly from \( vp(S_b^*) \) by virtue of the one-to-one correspondence between edges in \( G_b \) and vertices in \( S_b^* \). For the proof of assertions (ii) and (iii), consider the following. (a) There can be no replicas of vertices with degree lower than 2. (b) Vertices in \( G_b \) with a degree lower than 2 are not represented by any edges in \( S_b^* \). (c) Vertices \( u \) in \( G_b \) that possess a degree \( d(u) \geq 2 \) are represented in \( S_b^* \) through unique, edge-disjoint paths, each connecting the \( d(u) \) vertices in \( S_b^* \) corresponding to the edges incident to \( u \) in \( G_b \).

To prove (ii), we show that the number of replicas of any vertex \( u \) in \( ep(G_b) \) does not exceed the number of cut edges directly induced by \( u \) in \( vp(S_b^*) \). From (a) and (b), it trivially holds for vertices with a degree lower than 2. Assuming there are \( x > 0 \) replicas of \( u \) in \( ep(G_b) \), it implies that edges incident on \( u \) are distributed across \( x + 1 \) nonempty blocks. According to (c), \( u \) is uniquely represented by an edge-disjoint path connecting the vertices in \( S_b^* \) that correspond to the edges of \( u \) in \( G_b \). When the vertices of a connected (sub)graph are partitioned into \( x + 1 \) blocks, these blocks themselves are interconnected. Therefore, the edge-cut directly attributable to vertex \( u \) is at least \( x \), which completes the proof for (ii).

To prove (iii), we show how to build a valid CSPAC graph \( S_b^* \), such that \( cost(ep(G_b)) = cost(vp(S_b^*)) \). For vertices of \( G_b \) with a degree lower than 2, the equality trivially holds based on (a) and (b). For a vertex \( u \) in \( ep(G_b) \) with \( x > 0 \) replicas (edges distributed across \( x + 1 \) nonempty blocks), we create \( x + 1 \) independent paths connecting the vertices in \( S_b^* \) that represent the edges incident to \( u \) in \( G_b \). Vertices in each path are then assigned to a common block. As these paths are between vertices of the same block, they have no cut edges. Subsequently, these paths are interlinked to form a unified path. This introduces exactly \( x \) cut edges directly associated with vertex \( u \), thereby concluding the proof for condition (iii) and the overall proof.

▶ Theorem. For any vertex partition \( vp(\beta_b) \) of the maximal model \( \beta_b \) with edge-cut \( cost(vp(\beta_b)) \), there exists a corresponding edge partition \( ep(G_b) \) of the batch graph \( G_b \) that, when incorporated into the already partitioned section of the input graph \( G \), introduces a number \( \gamma \) of new vertex replicas, satisfying \( \gamma \leq cost(vp(\beta_b)) \).

Proof. The only disparity between the CSPAC graph \( S_b^* \) and the maximal model \( \beta_b \) is the presence of artificial vertices and edges in \( \beta_b \). If there are no artificial edges in \( \beta_b \), it implies that none of the edges in the batch graph \( G_b \) are adjacent to edges in \( G \) that have previously been assigned to blocks. In this scenario, Theorem 1 provides sufficient grounds to establish the claim. If \( \beta_b \) contains artificial edges, each artificial edge signifies a unique adjacency in \( G \) of an edge in the batch graph \( G_b \) to edges already assigned to blocks in previous batches. In this scenario, we complete the proof of the claim by noting that the number of cut artificial edges in \( vp(\beta_b) \) cannot be less than the number of new replicas introduced exclusively for vertices contained in previous batches.

▶
Faster Treewidth-Based Approximations for Wiener Index

Giovanna Kobus Conrado
Hong Kong University of Science and Technology (HKUST), Clear Water Bay, New Territories, Hong Kong

Amir Kafshdar Goharshady
Hong Kong University of Science and Technology (HKUST), Clear Water Bay, New Territories, Hong Kong

Pavel Hudec
Hong Kong University of Science and Technology (HKUST), Clear Water Bay, New Territories, Hong Kong

Pingjiang Li
Hong Kong University of Science and Technology (HKUST), Clear Water Bay, New Territories, Hong Kong

Harshit Jitendra Motwani
Department of Computer Science and Engineering & Department of Mathematics, Hong Kong University of Science and Technology (HKUST), Clear Water Bay, New Territories, Hong Kong

Abstract

The Wiener index of a graph $G$ is the sum of distances between all pairs of its vertices. It is a widely-used graph property in chemistry, initially introduced to examine the link between boiling points and structural properties of alkanes, which later found notable applications in drug design. Thus, computing or approximating the Wiener index of molecular graphs, i.e. graphs in which every vertex models an atom of a molecule and every edge models a bond, is of significant interest to the computational chemistry community.

In this work, we build upon the observation that molecular graphs are sparse and tree-like and focus on developing efficient algorithms parameterized by treewidth to approximate the Wiener index. We present a new randomized approximation algorithm using a combination of tree decompositions and centroid decompositions. Our algorithm approximates the Wiener index within any desired multiplicative factor $(1 \pm \epsilon)$ in time $O(n \cdot \log n \cdot k^3 + \sqrt{n} \cdot k/\epsilon^2)$, where $n$ is the number of vertices of the graph and $k$ is the treewidth. This time bound is almost-linear in $n$.

Finally, we provide experimental results over standard benchmark molecules from PubChem and the Protein Data Bank, showcasing the applicability and scalability of our approach on real-world chemical graphs and comparing it with previous methods.

2012 ACM Subject Classification Theory of computation → Parameterized complexity and exact algorithms

Keywords and phrases Computational Chemistry, Treewidth, Wiener Index

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.6

Funding The research was partially supported by the Hong Kong Research Grants Council ECS Project Number 26208122. G.K. Conrado and P. Hudec were supported by the Hong Kong PhD Fellowship Scheme (HKPFS).

Acknowledgements Authors are ordered alphabetically.
1 Introduction

Motivation
The Wiener index of a graph $G$ is the sum of the distances between all pairs of vertices in $G$. Besides being a natural problem to compute, it is also a well-studied graph invariant with applications in computational chemistry and biology. Indeed, it is one of computational chemistry’s oldest and most important topological indices [60].

History
In chemistry, the Wiener index was first considered by Harry Wiener in [63]. It was initially studied to establish connections between alkanes’ boiling points and the underlying graphs’ structural properties. This study later motivated the development of other topological indices in computational chemistry. Further development of QSAR (Quantitative Structure-Activity Relationship) and QSPR (Quantitative Structure-Property Relationship) models led to the discovery of positive correlations of even more chemical and physical properties to the Wiener index [48, 60, 61, 65]. Due to its simplicity and usefulness, the Wiener index was also studied by computer scientists and mathematicians [31, 57]. The use of neural networks in chemical graph theory has led to a renewed interest in topological indices and their application in molecular mining, toxicity detection, and computer-aided drug discovery. Several studies have been conducted on this topic, such as [10, 30, 32, 44, 64]. Given the significance of the Wiener index for chemists and the abundance of large molecules, it is imperative to develop faster algorithms for computing it. Indeed, there are many previous works in this direction [12, 22, 29, 40, 50].

Parameterized Algorithms
Parameterized algorithms aim to tackle computationally-intractable problems and identify subsets of instances that can be solved efficiently [26]. In parameterized complexity, we consider an additional parameter $k$ along with the input size $n$ for measuring the runtime. This is in contrast to classical complexity theory, which only considers the input size of the problem. Many parameterized algorithms focus on NP-hard problems and provide runtime bounds that depend polynomially on the size of the problem but have non-polynomial dependence on the parameter $k$. If we know that $k$ is small in real-world instances, this leads to solutions that are effectively polynomial-time, i.e. they take polynomial time on all the real-world instances where this parameter is small.

Fixed-Parameter Tractable (FPT)
Given an input of size $n$ and a parameter $k$, an algorithm with a running time of $O(f(k)\cdot n^c)$, for some constant $c$ and computable function $f$, is called Fixed-Parameter Tractable (FPT) [26]. The intuition is the same as above. If the parameter $k$ is small in all real-world instances of the problem, then the algorithm would in practice have a polynomial runtime. Crucially, the degree $c$ of this polynomial does not depend on either $k$ or $n$.

Treewidth
Treewidth is one of the most important structural parameters of graphs and has been extensively studied in combinatorics and graph theory. Intuitively speaking, it measures the tree-likeness of a graph [9]. Trees and forests have a treewidth of 1 and cliques on $n$ vertices have treewidth $n-1$. The main advantage of treewidth in algorithm design arises when we are designing parameterized algorithms for NP-hard problems by considering it as the
parameter of the problem. Many families of commonly-studied graphs, such as trees, cacti, series-parallel graphs, outer-planar graphs, control-flow graphs of structured programs, and conflict graphs of Bitcoin transactions have bounded treewidth [7, 9, 26, 18, 59, 13, 49, 25]. This allows efficient dynamic programming techniques using the tree decomposition of the graph [7, 37, 17, 3, 2, 39, 24]. See Section 2 for a formal definition.

Treewidth of Molecules

Extending this idea, computational chemists and biologists have also explored the treewidth of various important classes of molecules [66, 68]. In our experimental results (Section 4), we observe that a significant majority of molecules in the PubChem repository [34] have a treewidth of at most 10. Even large proteins from the Protein Data Bank [54] are observed to have a treewidth of at most 5. Since a significant fraction of molecules have bounded treewidth, exploring and designing treewidth-based parameterized algorithms for computational problems in chemistry and biology is a natural step. In fact, the same has been done in several works in the literature [4, 12, 23, 62, 67]. We extend this line of research by presenting significantly faster treewidth-based approaches for approximating the Wiener index.

Our Contribution

In this paper, we introduce a novel randomized algorithm that approximates the Wiener index of a graph using its tree decomposition. The unique aspect of our algorithm is the incorporation of both tree and centroid decompositions. This idea significantly enhances efficiency in answering distance queries within the graph. This is then plugged directly into an established randomized algorithm to approximate the Wiener index, obtaining the same approximation guarantees by an asymptotically faster method. Both theoretical analysis and experimental results demonstrate that our algorithm outperforms current methods in calculating the Wiener index for molecular graphs, which are commonly encountered in computational chemistry and biology.

Comparison with Previous Results

Table 1 compares the runtime complexity of our algorithm with previous methods. Here, \( n \) is the number of vertices in the graph, \( k \) is the treewidth, and \( \epsilon \) is the error in the approximation, i.e. we are reporting the runtime for a \((1 \pm \epsilon)\)-approximation of the Wiener index. We refer to Section 4 for a detailed experimental evaluation of our algorithm on datasets from PubChem [34] and the Protein Data Bank [54].

The most classical approach to compute the Wiener index is simply performing an all-pairs shortest path computation using Floyd-Warshall and then summing up the distances. This will lead to a time complexity of \( O(n^3) \). In [12], the authors provided the first parameterized algorithm for the Wiener index based on treewidth. Their algorithm is a divide-and-conquer method based on orthogonal range searching and repeatedly finds small cuts using the tree decomposition. They achieve a runtime bound of \( O(n \cdot \log^{k-1} n) \). Note that this is not FPT. In [21], an FPT algorithm was provided based on dynamic programming on the tree decomposition. This algorithm has a quadratic dependence on \( n \). For unweighted graphs, given that a graph with \( n \) vertices and treewidth \( k \) has \( O(n \cdot k) \) edges, running a BFS from each vertex would lead to a total runtime of \( O(n^2 \cdot k) \). Finally, [40] provides an algorithm on general graphs, not using any parameters, that approximates the average pairwise distance within a factor of \((1 \pm \epsilon)\) with a probability of at least \( 2/3 \) by taking a random sample of the distances between pairs of vertices. Note that the Wiener index is \( n^2 \) times the average
Table 1 Comparison of Different Algorithms for Computing the Wiener Index. Here, \( n \) denotes the number of vertices, \( k \) denotes the treewidth, and \( \epsilon \) represents the error of approximation.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Complexity</th>
<th>Type</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floyd–Warshall</td>
<td>( O(n^3) )</td>
<td>Exact</td>
<td>[33]</td>
</tr>
<tr>
<td>Orthogonal Range Searching</td>
<td>( O(n \cdot \log^{k-1} n) )</td>
<td>Exact, Parameterized</td>
<td>[12]</td>
</tr>
<tr>
<td>Treewidth-based Dynamic</td>
<td>( O(n^2 \cdot k^2) )</td>
<td>Exact, Parameterized</td>
<td>[21]</td>
</tr>
<tr>
<td>Programming</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFS</td>
<td>( O(n^2 \cdot k) )</td>
<td>Exact, Parameterized</td>
<td>[51, 69]</td>
</tr>
<tr>
<td>Classical Approximation</td>
<td>( O(n^{5/2}/\epsilon^2) )</td>
<td>Randomized, Approximation</td>
<td>[40]</td>
</tr>
<tr>
<td>Our Algorithm</td>
<td>( O(n \cdot \log n \cdot k^3 + \sqrt{n} \cdot k/\epsilon^2) )</td>
<td>Parameterized, Randomized, Approximation</td>
<td>Sec. 3</td>
</tr>
</tbody>
</table>

distance. Thus, this algorithm is directly applicable to our setting, as well. Our algorithm builds upon the classical approximation of [40] and uses a tree decomposition and a centroid decomposition to speed up the sampling.

Similar Works

Our distance query results are similar to those of [53, 41, 6, 1, 15, 19, 20, 14, 16]. However, unlike previous works that obtain a balanced tree decomposition, i.e. a tree decomposition with height \( O(\log n) \), our approach looks at the centroid decomposition of a tree decomposition. This centroid decomposition is not necessarily a valid tree decomposition of the original graph, but it has the same set of bags as the tree decomposition. Hence, unlike several previous works, our approach does not increase the width in order to obtain a balanced tree.

2 Preliminaries

In this section, we introduce the Wiener index and define some basic concepts of parameterized complexity. We refer to [26] for more details. This is followed by a short presentation of the classical approximation algorithm of [40], which forms the basis of our approach.

Wiener Index [63]

The Wiener Index of an undirected graph \( G = (V, E) \) is defined as the all-pairs sum of distances among vertices of the graph. Formally,

\[
W(G) := \sum_{u, v \in V} d(u, v).
\]

Additionally, we define the average distance between pairs of vertices in \( G \) as \( \bar{d}(G) := W(G)/n^2 \).

▶ Remark 1. In this work we assume that our graphs are connected, unweighted, and undirected. In the context of molecular graphs, all types of covalent bonds – be they single, double, or triple – are represented as a single undirected edge in the corresponding graph. For a disconnected graph, the Wiener index is simply \(+\infty\). However, in some applications, the Wiener index of a disconnected graph is defined as the sum of the Wiener indices of its connected components. In such cases, each connected component can be processed separately. Our algorithm can easily be extended to weighted graphs, as well.
Tree Decomposition (TD) [43, 55, 56]

A tree decomposition of a given graph \( G = (V,E_G) \) is a tree \( T = (B,E_T) \) satisfying the following conditions:

- Every node \( b \in B \) of \( T \), which is called a bag, contains a subset of vertices \( V_b \subseteq V \).
- The bags cover the entire vertex set \( V \) of \( G \), i.e. \( \bigcup_{b \in B} V_b = V \). In other words, every vertex appears in at least one bag.
- For every edge in the original graph \( G \), there exists a bag that contains both endpoints of the edge. More formally, for every \( e = \{u,v\} \in E_G \), there is a bag \( b \in B \), s.t. \( u, v \in V_b \).
- Every vertex \( v \in V \) appears in a connected subtree of \( T \), meaning that the set \( B_v = \{ b \in B \mid v \in V_b \} \) forms a connected subgraph of \( T \).

▶ Remark 2. An equivalent statement of the last condition above is that for every three bags \( b_1, b_2, b_3 \in B \), if \( b_3 \) is on the unique path from \( b_1 \) to \( b_2 \) in \( T \), then \( V_{b_1} \cap V_{b_2} \subseteq V_{b_3} \).

Treewidth [55]

The width of a tree decomposition \( T \) is defined as \( w(T) := \max_{b \in B} |V_b| - 1 \), i.e. the size of the largest bag minus one. Furthermore, the treewidth of the graph \( G \), denoted as \( tw(G) \), is defined as the minimum width amongst all possible tree decompositions of \( G \).

Intuitively speaking, treewidth measures the structural likeness of a graph to a tree. Specifically, the smaller the treewidth of a graph, the more tree-like it appears, in the sense that a graph of treewidth \( k \) can be decomposed into small parts (bags), each of size at most \( k + 1 \), which are connected to each other in a tree-like manner \( T \). Figure 1 showcases an illustration containing two distinct tree decompositions of a graph \( G \), each having a different width. Since only forests have treewidth of 1, the tree decomposition on the right is optimal, and \( tw(G) = 2 \).

![Figure 1](A Graph G and Two Tree Decompositions of G of Width 3 (left) and 2 (right).)

Treewidth is a parameter indicating graph sparsity, providing an upper bound on the number of edges. Specifically, in a graph with \( n \) vertices and treewidth \( k \), the number of edges is \( O(n \cdot k) \). More precisely, the number of edges is less than or equal to \( n \cdot k - k \cdot (k+1/2) \) [52]. Additionally, we have the following ubiquitous lemma:

▶ Lemma 3 (Cut Lemma [26]). Let \( T = (B,E_T) \) be a tree decomposition of \( G = (V,E_G) \). Consider two vertices \( u,v \in V \) and two arbitrary bags \( b_u, b_v \in B \) such that \( u \in b_u \) and \( v \in b_v \). If \( b \in B \) is a bag on the unique path from \( b_u \) to \( b_v \) in \( T \), then any path from \( u \) to \( v \) in \( G \) will intersect \( V_b \). Additionally, if \( e = \{b_1,b_2\} \in E_T \) is an edge on the unique path from \( b_u \) to \( b_v \) in \( T \), then any path from \( u \) to \( v \) in \( G \) will intersect \( V_{b_1} \cap V_{b_2} \).
Computing Tree Decompositions

In general, computing the treewidth of a given graph is an NP-hard problem. However, for small values of \( k \), it is well-known that we can decide whether the treewidth of a given graph is at most \( k \) and also compute as an optimal tree decomposition with \( O(n) \) bags by a linear-time FPT algorithm (parameterized by the treewidth itself and depending exponentially on \( k \)) \cite{8}. Additionally, there are many well-optimized tools for this task. Thus, in the sequel, we assume without loss of generality that an optimal tree decomposition of our graph is given as a part of the input.

Centroid \cite{45}

Consider a tree \( T = (V_T, E_T) \) with \( n \) vertices. We define a centroid node of \( T \) as a node whose removal breaks the tree down into several subtrees such that no resulting subtree has a size greater than \( n/2 \). In other words, a centroid is a 1/2-separator of \( T \). It is well-known that every tree has at least one centroid node, which can be obtained in linear time by dynamic programming.

Centroid Decomposition (CD) \cite{11, 27}

A centroid decomposition of \( T \) is another tree \( T' \) on the same set of vertices as \( T \), recursively defined as follows:

- When \( |V_T| = 1 \), we simply have \( T' = T \).
- For a more complex tree, we first identify a centroid node \( r \) of \( T \), then position this node as the root of \( T' \).
- Once we have selected a centroid node \( r \) and removed it from \( T \), we end up separating the original tree into several connected subtrees. Let us denote these as \( T_1, T_2, \ldots, T_m \). For each subtree \( T_i \), we find a centroid decomposition \( T'_i \) with a root \( r_i \). We make each \( r_i \) a child of \( r \).

Figure 2 shows the steps of computing a centroid decomposition. Each color corresponds to a distinct layer of the centroid decomposition, with the node representing the centroid of the similarly colored dotted subtree. In this illustration, the node 4 is identified as the centroid of the initial tree. Following the removal of node 4, nodes 2, 7, and 12 are selected as the centroids of each resulting subtree. Subsequent centroids are determined in a recursive manner. The final centroid decomposition is shown in Figure 3.

\[ \text{Figure 2} \quad \text{A Graph} \ G \ \text{and the Steps of Building its Centroid Decomposition. Each step highlights the centroid vertex of each of the current components of the graph.} \]
Properties of CDs

The height of a CD is bounded by $O(\log n)$, where $n$ is the number of vertices in the original tree. This is because with every new layer added to the centroid decomposition, each connected component splits into several parts, each no larger than $1/2$ the size of the original component. Consequently, we can append at most $O(\log n)$ layers to the centroid decomposition. Additionally, CDs satisfy the following useful lemma:

▶ Lemma 4 (Proof in Appendix A). Let $u, v \in V_T$ be two vertices of the original tree $T$ and $l$ be their lowest common ancestor in the centroid decomposition $T'$. The unique path connecting $u$ and $v$ in $T$ must visit $l$.

Computing Centroid Decompositions

Given a tree $T$ with $n$ vertices, there are a variety of algorithms in the literature that compute a centroid decomposition $T'$ of $T$ in $O(n)$. Examples include [11, 27].

Lowest Common Ancestor Queries

Consider a rooted tree $T$ with $n$ vertices. Suppose we have $q$ offline queries, each providing two vertices $u, v \in T$ and asking for their lowest common ancestor. The classical algorithm of Gabow and Tarjan [35] solves this problem and answers all queries in $O(n + q)$.

Approximation Algorithm of [40]

The work [40] provides an elegant and simple approximation algorithm for the average distance $\overline{d}(G)$ between pairs of vertices. Since the Wiener index is simply $n^2 \cdot \overline{d}(G)$, the same algorithm can be reused for our problem. Given a graph $G$ and an error bound $\epsilon$ as the input, the algorithm in [40] works as follows:

1. Uniformly select $\Theta(\sqrt{n}/\epsilon^2)$ pairs of vertices.
2. Find the distance between each selected pair of vertices.
3. Output the average of the computed distances.

Surprisingly, this algorithm provides a $(1 \pm \epsilon)$-approximation of $\overline{d}(G)$ with probability $2/3$.

▶ Theorem 5 ([40], Theorem 5.1). Given $G$ and $\epsilon$ as input, the algorithm above outputs a $(1 \pm \epsilon)$-approximation of $\overline{d}(G)$ with probability at least $2/3$.

As a direct corollary, a $(1 \pm \epsilon)$-approximation of the Wiener index can be computed in the same time complexity by simply multiplying the result of this algorithm by $n^2$. 
Complexity Analysis

For general graphs, each distance query can take \(O(n^2)\) time. Thus, the total runtime of the algorithm above is \(O(n^{3/2}/\epsilon^2)\). However, if the underlying graph \(G\) is guaranteed to have small treewidth \(k\), then it can have at most \(O(n \cdot k)\) edges. Thus, each distance query can be answered in \(O(n \cdot k)\) by a BFS. This reduces the runtime to \(O(n^{3/2} \cdot k/\epsilon^2)\).

In this work, we build upon this simple and classical randomized algorithm and use the treewidth to obtain a faster algorithm for distance queries. This allows us to reduce the runtime dependence on \(n\) to almost-linear.

3 Our Algorithm

In this section, we present our treewidth-based algorithm. Our algorithm follows the same steps as the approximation algorithm of [40], except that we exploit the tree decomposition to perform distance queries faster. Our main novel idea is to look not only at a tree decomposition of the underlying graph but also at a centroid decomposition of this tree decomposition. Thus, our algorithm exploits the desirable properties of both types of decomposition, as formalized by the lemma below:

Lemma 6. Let \(G = (V, E_G)\) be a graph, \(T = (B, E_T)\) a tree decomposition of \(G\) and \(T' = (B, E_{T'})\) a centroid decomposition of \(T\). Consider two vertices \(u, v \in V\) and arbitrary bags \(b_u, b_v \in B\) such that \(u \in b_u\) and \(v \in b_v\). Let \(l\) be the lowest common ancestor of \(b_u\) and \(b_v\) in the centroid decomposition \(T'\). Any path that goes from \(u\) to \(v\) in \(G\) intersects \(V_l\).

Proof. Consider a path \(\pi_T\) from \(b_u\) to \(b_v\) in the tree decomposition \(T\). By Lemma 4, we have \(l \in \pi_T\). By Lemma 3, any bag in \(\pi_T\) intersects every path from \(u\) to \(v\) in \(G\). This is illustrated in Figure 4.

Based on the lemma above, if we precompute the distances from each vertex appearing in a bag \(l\) of the centroid decomposition \(T'\) to the vertices appearing in descendants of \(l\) in \(T'\), then we can answer distance queries in \(O(k)\). In other words, to find the distance from \(u\) to \(v\), we first find two bags \(b_u\) and \(b_v\) containing them, then compute \(l = \text{lca}(b_u, b_v)\). Now, we know that every path from \(u\) to \(v\) has to go through \(l\), thus

\[
    d_G(u, v) = \min_{w \in V_l} \left( d_G(u, w) + d_G(w, v) \right).
\]

Here, \(d_G\) denotes the distance in our graph \(G\).

Our Algorithm for Wiener Index

Based on the discussion above, given \(\epsilon > 0\), a graph \(G = (V, E_G)\) and a tree decomposition \(T = (B, E_T)\) of \(G\) with width \(k\), our algorithm turns \(G\) into a weighted graph and takes the following steps:

1. **Step 1 (Centroid Decomposition).** Compute a centroid decomposition \(T'\) of the tree decomposition \(T\).
2. **Step 2 (Local Precomputation).** For every two vertices \(u, v \in V\), if there is a bag \(b \in B\) that contains both of them, i.e. \(u, v \in V_b\), then compute the distance \(d_G(u, v)\) and add a direct edge with weight \(d_G(u, v)\) between \(u\) and \(v\).
3. **Step 3 (Ancestor-Descendant Precomputation).** Let \(b_1, b_2 \in B\) be two bags such that \(b_1\) is an ancestor of \(b_2\) in the centroid decomposition \(T'\). For every \(u \in V_{b_1}\) and \(v \in V_{b_2}\), compute the distance \(d_G(u, v)\) and add a direct edge with weight \(d_G(u, v)\) between \(u\) and \(v\).
Step 4 (Sampling). Uniformly select $\Theta(\sqrt{n}/\epsilon^2)$ pairs of vertices of $G$ as in the algorithm of [40].

Step 5 (Distance Queries). For each pair of vertices $(u, v) \in V^2$ selected in the previous step, compute $d_G(u, v)$.

Step 6 (Output). Output the average of all the distances obtained in the previous step.

For Step 1, we can rely on previous algorithms that compute centroid decompositions, such as [11, 27]. Steps 4 and 6 are straightforward. We now provide details of Steps 2, 3, and 5, followed by correctness proofs and runtime analyses.

Details of Step 2

This step is inspired by and similar to [21, 5, 36, 38]. Given the graph $G = (V, E_G)$ and its tree decomposition $T = (B, E_T)$, our goal is to create shortcut edges between any pair of vertices that appear in the same bag. We provide a recursive procedure as follows:

i. Choose a leaf bag $\ell$ of the tree decomposition $T$.

ii. Perform an all-pairs shortest-path algorithm, such as Floyd-Warshall, in $G[V_\ell]$, i.e. only on the vertices and edges in $\ell$. If a path of length $d$ is found between $u$ and $v$, add a direct $\{u, v\}$ edge with weight $d$ to $G$.

iii. Let $T^* = T - \ell$ and $G^* = G - \{v \in V_\ell \mid \exists b \in B \ b \neq \ell \land v \in V_b\}$. In other words, we are removing the leaf bag $\ell$ from our tree decomposition and also removing any vertex that appeared only in this bag from the graph $G$.

iv. Run the algorithm recursively on $(G^*, T^*)$. This causes more shortcut edges to be added in $G$.

v. Repeat Step ii, i.e. perform another all-pairs shortest-path in $G[V_\ell]$ and add the resulting shortcut edges to $G$.

Figure 5 provides an example of this step.
We choose the leaf bag $\ell = \{3, 8, 9\}$.

Shortest paths are found within $G[\mathcal{V}_{\ell}]$. Dashed lines represent newly added edges.

Bag $\ell$ is removed from $T$ to form $T^\ast$.

$G^\ast$ is formed by removing vertex 9 from $G$, since it only appears in bag $\ell$ in $T$.

After recursively running the algorithm on $(G^\ast, T^\ast)$, new edges are added to $G$.

Shortest paths are again found within $G[\mathcal{V}_{\ell}]$ and any new edges found are added to $G$. In this example, no new edges were found.

Figure 5 An Example of Step 2 on the Graph and Decomposition of Figure 4.

Lemma 7 (Proof in Appendix B). The procedure above runs in time $O(n \cdot k^3)$. After its execution, $T$ is still a valid tree decomposition of $G$, and for every pair of vertices $u, v \in V$, if there exists a bag $b \in \mathcal{B}$ containing both of them, then there is a direct (shortcut) edge from $u$ to $v$ with weight $d_G(u, v)$.

Remark 8. Throughout our algorithm, we always keep at most one edge, i.e. the edge with minimum weight, between every pair \{u, v\} of vertices.

Details of Step 3

In this step, we process our centroid decomposition $T'$ in a bottom-up manner. For every bag $b \in \mathcal{B}$, we consider the subtree $T'_b$ of the centroid decomposition $T'$, consisting of $b$ and all of its descendants in $T'$. Let $G_b$ be the induced subgraph of $G$ that contains all the vertices in $T'_b$, i.e.

$$G_b = G \left[ \bigcup_{b' \in T'_b} V_{b'} \right].$$

For every vertex $v \in V_b$ that appears in the bag $b$, our algorithm runs a shortest-path computation, such as Dijkstra’s algorithm [28], from $b$ in the graph $G_b$ and finds its distances to all other vertices of $G_b$, adding the corresponding shortcut edges. See Figure 6 for an example.

Lemma 9. The procedure above runs in $O(n \cdot \log n \cdot k^3)$ time. After its execution, for every two bags $b_1, b_2 \in \mathcal{B}$ such that $b_1$ is an ancestor of $b_2$ in the centroid decomposition $T'$ and every two vertices $u \in V_{b_1}$ and $v \in V_{b_2}$, we have a shortcut edge from $u$ to $v$ with weight $d_G(u, v)$. 
The latter equality is because every vertex has \( \delta_b \) and \( \alpha_b \) be the number of ancestors and descendants of \( b \) in \( T' \), respectively. The graph \( G_b \) has \( O(\delta_b \cdot k) \) vertices and thus \( O(\delta_b \cdot k^2) \) edges. Moreover, we perform \( O(k) \) Dijkstras over this graph, one for each vertex in the bag \( b \). Our graph is weighted at this point, but all edge weights and distances are non-negative integers less than \( n \). Thus, Dijkstra runs in linear time on the number of vertices and edges. Intuitively, instead of keeping a priority queue of vertices in our Dijkstra, we can simply keep an array \( A[i] \) of queues where \( A[i] \) contains all vertices of distance \( i \) to the source. When we find that a particular vertex has distance \( i \) to the source, we simply add it to \( A[i] \). We then process the vertices in each \( A[i] \) in the order of increasing \( i \) and make sure not to process a vertex more than once.

Based on the points above, our total runtime is

\[
\sum_{b \in B} O(\delta_b \cdot k^3) = \sum_{b \in B} O(\alpha_b \cdot k^3) = O(n \cdot \log n \cdot k^3).
\]

The latter equality is because every vertex has \( O(\log n) \) ancestors.

For the second part, consider a shortest path \( \pi \) from \( u \) to \( v \) in \( G \). Let \( \pi_T \) be the path from \( b_1 \) to \( b_2 \) in the tree decomposition \( T \). By Lemma 3, \( \pi \) intersects the vertices of every bag \( b \) in \( \pi_T \). Without loss of generality, we can assume that \( \pi \) stays in these bags, i.e. it only visits vertices in \( \bigcup_{b \in T} \mathcal{V}_b \) where \( \mathcal{V}_b \) is a bag in the same bag and, by Lemma 7, there is already a shortcut edge between them. Additionally, since \( b_1 \) is an ancestor of \( b_2 \), there was a point in the construction of \( T' \) when \( b_1 \) was chosen as the centroid of a connected component containing \( b_2 \). Thus, the entire path \( \pi \) is included in \( G_b \) and the Dijkstra from \( u \) finds the shortest path to \( v \) and adds the corresponding shortcut edge.

**Details of Step 5**

Suppose our goal is to compute \( d_G(u, v) \). We first pick two bags \( b_u \) and \( b_v \) such that \( u \in b_u \) and \( v \in b_v \). We then find the lowest common ancestor \( l = \text{lca}(b_u, b_v) \). By Lemma 4, every path from \( u \) to \( v \) has to intersect \( \mathcal{V}_l \). Thus, we compute

\[
d_G(u, v) = \min_{w \in \mathcal{V}_l} (d_G(u, w) + d_G(w, v)).
\]

Note that since \( l \) is an ancestor of both \( b_u \) and \( b_v \), we have the distances needed on the RHS as weights of direct shortcut edges. This is illustrated in Figure 7

**Lemma 10.** The procedure above returns the correct distances in time \( O(n + k \cdot \sqrt{n}/\epsilon^2) \).
(a) Let \( u = 5 \) and \( v = 7 \).

(b) We select \( b_u = \{3, 4, 5\} \) and \( b_v = \{3, 7, 8\} \). Their lowest common ancestor \( l \) in \( T' \) is \( \{1, 2, 3\} \), thus any shortest path from 5 to 7 has to go through vertices 1, 2, or 3.

(c) Since \( l \) is an ancestor of \( b_u \) and \( b_v \), the vertices \( u \) and \( v \) have shortcut edges to every vertex in \( l \).

Figure 7 An Example of Step 5 on the Graph and Decompositions of Figure 4.

Proof. Correctness is already argued above. Since the centroid decomposition \( T' \) has \( O(n) \) bags, preprocessing and answering offline lowest common ancestor queries takes \( O(n + \sqrt{n}/\epsilon^2) \) [35]. For each of the \( \sqrt{n}/\epsilon^2 \) queries generated in Step 4, we should compute the minimum of \( O(k) \) values since \( |V_l| \leq k + 1 \).

Finally, the following is our main theorem in this work:

\[ \textbf{Theorem 11.} \] Given an \( \epsilon > 0 \), an undirected unweighted graph \( G = (V, E_G) \) with \( n \) vertices and a tree decomposition \( T = (B, E_T) \) of \( G \) with \( O(n) \) bags and width \( k \), our algorithm runs in time \( O(n \cdot \log n \cdot k^3 + \sqrt{n} \cdot k/\epsilon^2) \) and produces a \((1 \pm \epsilon)\)-approximation of the Wiener index \( W(G) \) with probability at least \( 2/3 \).

Proof. Correctness of the approximation ratio and success probability follows from Theorem 5 since our algorithm is the same as [40] except for how we answer distance queries. Step 1 takes \( O(n) \) using well-known algorithms such as [11, 27]. Step 2 takes \( O(n \cdot k^3) \) based on Lemma 7. Step 3 takes \( O(n \cdot \log n \cdot k^3) \) as shown in Lemma 9. Step 4 simply takes \( O(\sqrt{n}/\epsilon^2) \) samples from the uniform distribution and Step 5 takes \( O(n + k \cdot \sqrt{n}/\epsilon^2) \) time as per Lemma 10. Finally, Step 6 takes \( O(\sqrt{n}/\epsilon^2) \) time. Summing these up leads to the desired asymptotic time complexity.

4 Experimental Results

In this section, we present our experimental results, comparing the runtimes of our algorithm with previous approaches. We implemented the main algorithms in C++ and provided the same inputs, i.e. graph \( G \), tree decomposition \( T \) and \( \epsilon = 0.1 \) to all of them. To obtain this input, we first used pysmiles [46], RDKit [47] and NetworkX [42] for preprocessing molecular data and turning them into graphs. We employed the FlowCutter algorithm [58] for tree decompositions, limiting iterations to \( 20 + \log n \), and obtained results in under 1 second. All our experiments were conducted on an Intel Core i5 (2.3 GHz, Quad-core) Machine with 8 GB of RAM running MacOS. We enforced a time limit of 1000 seconds per instance.

Benchmarks

We used the following datasets for our experiments: (i) PubChem [34] and (ii) Protein Data Bank (PDB). Specifically, we report results on 1049 randomly-selected protein molecules from the PDB database and 1,311,229 molecules from PubChem.
Table 2 Statistics of the PDB Benchmarks.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Vertices</td>
<td>132</td>
<td>90507</td>
<td>6651</td>
</tr>
<tr>
<td>Number of Edges</td>
<td>134</td>
<td>98828</td>
<td>6820</td>
</tr>
<tr>
<td>Treewidth</td>
<td>2</td>
<td>5</td>
<td>3.13</td>
</tr>
</tbody>
</table>

Table 3 Statistics of the PubChem Benchmarks.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Vertices</td>
<td>2</td>
<td>568</td>
<td>21</td>
</tr>
<tr>
<td>Number of Edges</td>
<td>1</td>
<td>643</td>
<td>22</td>
</tr>
<tr>
<td>Treewidth</td>
<td>1</td>
<td>16</td>
<td>1.8</td>
</tr>
</tbody>
</table>

PDB

The Protein Data Bank (PDB) [54] is an extensive repository of three-dimensional structural data for large biological molecules, including proteins, DNA and RNA. We randomly selected 1049 protein molecules from this database. Table 2 shows some statistics about these molecules. We observed that even the large molecules in this dataset have small treewidth.

PubChem


Treewidth of the Molecules

As mentioned in Tables 2 and 3, we observed that the chemical compounds in both benchmark suites exhibit small treewidth. Figure 8 provides a histogram for each benchmark suite. Notably, the vast majority of PubChem compounds have a treewidth of less than 10, with very few molecules having treewidths of up to 16. In addition, the large molecules in the PDB dataset also have bounded treewidths of at most 5.

Results

Figure 9 compares the performance of our algorithm and previous methods over the PDB dataset, whereas Table 4 provides the same comparison for PubChem. Our approach’s better asymptotic complexity leads to significant gains in efficiency when considering the large graphs in PDB. However, no benefit is observed over the PubChem molecules, since they are all small and every algorithm can handle them in under 1 ms.
Faster Treewidth-Based Approximations for Wiener Index

Figure 8  Treewidth Distribution in Our Benchmarks.

Figure 9  Runtime Comparison of the Algorithms of Table 1 over PDB Benchmarks. Each dot corresponds to one benchmark molecule.

Table 4  Runtime Comparison of the Algorithms of Table 1 over PubChem Benchmarks. All times are in milliseconds.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Maximum</th>
<th>Minimum</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our Algorithm</td>
<td>1.425</td>
<td>0.187</td>
<td>0.296454</td>
</tr>
<tr>
<td>Approximation Algorithm</td>
<td>1.283</td>
<td>0.203</td>
<td>0.297342</td>
</tr>
<tr>
<td>DP on Tree Decomposition</td>
<td>1.256</td>
<td>0.192</td>
<td>0.296152</td>
</tr>
<tr>
<td>Floyd-Warshall</td>
<td>2.261</td>
<td>0.199</td>
<td>0.288638</td>
</tr>
<tr>
<td>Orthogonal Range Searching</td>
<td>1.121</td>
<td>0.199</td>
<td>0.292404</td>
</tr>
<tr>
<td>BFS</td>
<td>1.097</td>
<td>0.205</td>
<td>0.290523</td>
</tr>
</tbody>
</table>

5 Conclusion

We considered the problem of computing the Wiener index, i.e. sum of all pairwise vertex distances, of a graph with \( n \) vertices and treewidth \( k \). We provided a novel algorithm using a combination of tree decompositions and centroid decompositions, which achieves an almost-
linear FPT runtime of $O(n \cdot \log n \cdot k^3 + \sqrt{n} \cdot k^2 \epsilon^2)$ and outputs a $(1 \pm \epsilon)$-approximation of the Wiener index with probability at least $2/3$. To our knowledge, this is the first sub-quadratic time FPT algorithm for this problem. We also showed that many real-world molecular graphs have small treewidth and thus our algorithm is applicable in practice.

References

Faster Treewidth-Based Approximations for Wiener Index


Thus, if \( \pi \) is a common ancestor. The remaining nodes on the path from \( u \) to \( v \) must traverse the node \( c \) as the remaining nodes on the path from \( u \) to \( v \) are either in \( T_i \) or \( T_j \) and, hence, cannot be a common ancestor.

**A  Proof of Lemma 4**

**Proof.** We prove this lemma through induction on the size \( n \) of the tree. If \( n \) is at most 3, the lemma holds trivially. Now assume that the lemma holds for all trees with a size less than \( n \). Let us consider a general tree of size \( n \). In the first step, we identify a centroid node of \( T \), denoted as \( c \). Removing \( c \) breaks \( T \) into several connected components. If any two vertices \( u, v \in V_T \) are in the same connected component \( T_\ell \), then the corresponding centroid decomposition \( T' \), they will appear in \( T'_\ell \) as per the definition of centroid decomposition. By the induction hypothesis, their path must cross their lowest common ancestor in \( T'_\ell \). In case they belong to different connected components, say \( T'_i \) and \( T'_j \), any path from \( T'_i \) and to \( T'_j \) must traverse the node \( c \). In this scenario, their lowest common ancestor would be the root \( c \), as the remaining nodes on the path from \( u \) to \( v \) are either in \( T_i \) or \( T_j \) and, hence, cannot be a common ancestor.

**B  Proof of Lemma 7**

**Proof.** We run the Floyd-Warshall algorithm twice on each bag of the tree decomposition, once in Step ii and once in v. Since each bag has \( k + 1 \) vertices and the tree decomposition has \( O(n) \) bags, the total runtime is \( O(n \cdot k^3) \). The procedure above adds new shortcut edges only between pairs of vertices that were already in the same bag, thus the tree decomposition remains valid.

We prove the last property by induction on \( |\mathcal{B}| \). If \( |\mathcal{B}| = 1 \), then the first Floyd-Warshall in Step ii adds all the necessary shortcut edges. Otherwise, let \( u, v \in V_\ell \) be two vertices that appear in the leaf bag \( \ell \) and let \( p \in \mathcal{B} \) be the parent of \( \ell \) in \( T_i \). If there is a path between \( u \) and \( v \) that is entirely within \( V_\ell \), then Step ii adds a shortcut edge summarizing this path. Thus, if \( u', v' \in G^* \), then \( d_{G^*}(u, v) = d_G(u, v) \). Moreover, both \( u \) and \( v \) have to appear in \( p \), since they each appear in a connected subtree. Hence, by induction hypothesis, the recursive call in Step iv adds the required shortcut edge between \( u \) and \( v \). Now consider the case where either \( u \) or \( v \) (or both) are not in \( G^* \). Take a shortest path \( \pi \) from \( u \) to \( v \) in \( G \). If \( \pi \) is entirely within \( V_\ell \), then Step ii adds the shortcut edge. Otherwise, we use Lemma 3 to break \( \pi \) down as \( \pi = \pi_1 \cdot w_1 \cdot \pi_2 \) where \( \pi_1 \) is the longest prefix of \( \pi \) that only contains vertices from \( V_\ell \setminus V_p \) and \( \pi_2 \) is the longest such suffix. By Lemma 3, we have \( w_1, w_2 \in V_\ell \cap V_p \). Since they

![Figure 10 An Illustration of Lemma 7.](image-url)
are both in $V_p \subseteq V_{G^*}$, Step iv adds a shortcut edge from $w_1$ to $w_2$. Hence, Step v adds a shortcut edge from $u$ to $v$ with the correct weight. Finally, if $u$ and $v$ are vertices that appear in the same bag $b \neq \ell$, then the recursive call on $(G^*, T^*)$ adds a shortcut edge between them.
Local Search $k$-means++ with Foresight

Theo Conrads
Department of Computer Science, University of Cologne, Germany

Lukas Drexler
Faculty of Mathematics and Natural Sciences, Department of Computer Science, Heinrich Heine University Düsseldorf, Germany

Joshua Könen
Institute of Computer Science, University of Bonn, Germany

Daniel R. Schmidt
Faculty of Mathematics and Natural Sciences, Department of Computer Science, Heinrich Heine University Düsseldorf, Germany

Melanie Schmidt
Faculty of Mathematics and Natural Sciences, Department of Computer Science, Heinrich Heine University Düsseldorf, Germany

Abstract

Since its introduction in 1957, Lloyd’s algorithm for $k$-means clustering has been extensively studied and has undergone several improvements. While in its original form it does not guarantee any approximation factor at all, Arthur and Vassilvitskii (SODA 2007) proposed $k$-means++ which enhances Lloyd’s algorithm by a seeding method which guarantees a $O(\log k)$-approximation in expectation. More recently, Lattanzi and Sohler (ICML 2019) proposed $LS++$ which further improves the solution quality of $k$-means++ by local search techniques to obtain a $O(1)$-approximation. On the practical side, the greedy variant of $k$-means++ is often used although its worst-case behaviour is provably worse than for the standard $k$-means++ variant.

We investigate how to improve $LS++$ further in practice. We study two options for improving the practical performance: (a) Combining $LS++$ with greedy $k$-means++ instead of $k$-means++, and (b) Improving $LS++$ by better entangling it with Lloyd’s algorithm. Option (a) worsens the theoretical guarantees of $k$-means++ but improves the practical quality also in combination with $LS++$ as we confirm in our experiments. Option (b) is our new algorithm, Foresight $LS++$. We experimentally show that $FLS++$ improves upon the solution quality of $LS++$. It retains its asymptotic runtime and its worst-case approximation bounds.

2012 ACM Subject Classification Mathematics of computing → Combinatorial algorithms; Theory of computation → Facility location and clustering; Information systems → Clustering

Keywords and phrases $k$-means clustering, $k$means++, greedy, local search

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.7

Supplementary Material Software: https://github.com/algo-hhu/FLSpp
archived at swh:1:dir:39136777e542456572a51515813b6cb77ed0940

Funding Lukas Drexler: Gefördert durch die Deutsche Forschungsgemeinschaft (DFG) – Projekt­nummer 456558332 – funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Projektnummer 456558332.
Joshua Könen: funded by Lamarr Institute for Machine Learning and Artificial Intelligence – lamarr-institute.org
Melanie Schmidt: Gefördert durch die Deutsche Forschungsgemeinschaft (DFG) – Projekt­nummer 456558332 – funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Projektnummer 456558332.

© Theo Conrads, Lukas Drexler, Joshua Könen, Daniel R. Schmidt, and Melanie Schmidt; licensed under Creative Commons License CC-BY 4.0

22nd International Symposium on Experimental Algorithms (SEA 2024).
Editor: Leo Liberti; Article No. 7; pp. 7:1–7:20
Leibniz International Proceedings in Informatics
Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


## 1 Introduction

In the vast area of clustering, one of the most popular approaches is min-sum-of-squared-error clustering, as modeled by the $k$-means cost function. Given a set of vectors $X = \{x_1, \ldots, x_n\} \subset \mathbb{R}^d$ and a number of clusters $k \in \mathbb{N}$, the aim of $k$-means clustering is to find a set $C = \{c_1, \ldots, c_k\}$ of $k$ centers that minimizes $\sum_{i=1}^{n} \min_{j=1,\ldots,k} ||x_i - c_j||^2$, i.e., the sum of the squared distances of all points to their respective closest center in $C$. For half a century, the most known algorithm for minimizing the $k$-means cost function was a local search heuristic often called $k$-means algorithm or Lloyd’s algorithm, developed in 1957. The main steps of the algorithm are as follows:

1. Find an initial set of $k$ centers, e.g., randomly chosen from $X$.
2. Repeat a given number $s$ times or until convergence:
   a. For every $x \in X$, find a closest center $\phi(x)$ (ties broken arbitrarily) and use this information to form clusters $C_1, \ldots, C_k$ by $C_i = \phi^{-1}(c_i)$.
   b. For all $C_i$, compute the mean $\mu(C_i) = \frac{1}{|C_i|} \sum_{x \in C_i} x$ and then replace $C$ by $C_{\text{new}} = \{\mu(C_1), \ldots, \mu(C_k)\}$.

Notice that this description ignores edge cases like clusters which “run empty” during step 2(a). We also do not discuss variations of other stopping criteria here. It is well documented [7, 15, 26] that the implementation of the first step of Lloyd’s algorithm is crucial because bad initial centers can lead to bad local optima. It is also easy to find theoretical worst case examples where the local optimum is arbitrarily bad [20].

In 2007, Arthur and Vassilvitskii [4] proposed a method to significantly improve the first step of Lloyd’s algorithm, leading to a new de-facto-standard algorithm for the $k$-means problem. The core idea is to choose the initial centers by an adaptive sampling procedure known as $d^2$-sampling initialization (also see Algorithm 1 below):

1. Choose $c_1$ uniformly at random from $X$.
2. For $i = 2, \ldots, k$:
   a. For all $x \in X$, compute $p(x) := \frac{\min_{y \in C} ||x-y||^2}{\sum_{x \in X} \min_{y \in C} ||y-x||^2}$
   b. Sample a point $c_i$ where every $x \in X$ has probability $p(x)$
3. Return $C = \{c_1, \ldots, c_k\}$

The algorithm proposed in [4] first computes $C$ by this routine and then runs Lloyd’s algorithm with $C$ as the initial center set. This combination of $d^2$-sampling and Lloyd’s algorithm is called $k$-means++.

Let us consider some background to better understand the advantages of $k$-means++ (and the subsequent improvements): There are two main reasons why a clustering obtained with Lloyd’s algorithm may be bad. Firstly, it may be that the underlying structure of the data does not fit the $k$-means objective, e.g., because the points are not well-clusterable with spherical clusters, or because we chose the wrong $k$. In this case, we should choose a different clustering method like kernel $k$-means, hierarchical clustering methods or density based clustering. But secondly, it may be that although optimal clusters with respect to the $k$-means objective are indeed perfect for us, Lloyd’s algorithm does not find them because it converges to a local optimum. Figure 1 gives a visual example for this: We see a $k$-means based image compression with four colors (The input pixels are represented by three-dimensional points based on the RGB-values, these points are clustered, and then the color of every pixel is replaced by the mean color of its cluster. Clusters are not contiguous within the picture). In this example, we can see how the difference in $k$-means cost is indeed reflected by the different quality of the compressed image.
Compression of an image with $k = 4$ centers (i.e. colors). Subfigure (a) shows the original image. Subfigure (b) shows a local optimum with a $k$-means cost of $55.18 \cdot 10^8$. We found this local optimum in runs of Lloyd with uniform initialization and in single runs of $k$-means++. Subfigure (c) shows a solution with a $k$-means cost of $43.09 \cdot 10^8$ (for example found by FLS++).

This data set is by Fritzke [16], the illustration by Conrads [9]. The left side shows nine centers sampled by one run of $k$-means++ and the corresponding induced clusters are illustrated by colors. The right side shows how the clusters and centers look after running Lloyd’s algorithm to convergence with the nine centers from the left as input.

Provided that the cost can actually guide us to a good solution, we know that there are hidden optimal clusters which we desire to find, and that we should improve upon the $k$-means cost to find them. The idea of $k$-means++ is to find one point from each optimal cluster. This point should be “relatively” good, in the sense that it should allow Lloyd’s algorithm to find a good center for the optimum cluster later. We call an optimum cluster covered when the $d^2$-sampling sampled one of its points. Now, observe that clusters that are not covered induce a high cost. By basing the $d^2$-sampling probabilities on the cost of points, the algorithm strives to find good points from uncovered optimum clusters. Indeed, $k$-means++ succeeds with this goal most of the time, but it can miss some of the optimum clusters – which then remain uncovered because the subsequent Lloyd steps are (in general) unable to shift a mean point from one optimum cluster to another. However, $k$-means++ still computes a $\Theta(\log k)$-approximation on expectation, and it improves the practical performance of Lloyd’s algorithm vastly. Figure 2 illustrates how $k$-means++ can end up in a local optimum. We see that out of the nine natural clusters in the data set, $k$-means++ has covered eight – the upper middle cluster was hit by two samples, and thus there was no center left in the end to cover the lower right cluster. Running Lloyd’s algorithm on this center set then results in nicely placed centers which capture the natural clusters perfectly in seven of nine cases. However, Lloyd’s algorithm cannot repair the mistake of not finding the lower right cluster.
It is now around fifteen years after the development of $k$-means++, and there has been increasing interest in the question of improving upon its theoretical and practical properties. There are two main ways to improve the behaviour explained in Figure 2. Option A is to try to avoid missing clusters. It was already proposed in [4] to improve $k$-means++ by adding a greedy procedure: When choosing $c_i$, compute $\ell$ candidates, all chosen according to the same probability distribution, and then out of these, pick the point which decreases the current cost the most. This variant of $k$-means++ is called greedy $k$-means++. Although it has been observed as performing better in practice in some instances (as in [4] and in [7]), greedy $k$-means++ actually has a worse worst-case behavior than $k$-means++. Bhattacharya et al. [6] give a family of point sets where the expected quality of solutions computed by greedy $k$-means++ is lower bounded by $\Omega(\ell \log k)$. This is because points which look beneficial to the greedy procedure may actually be very bad centers, for instance, if points lie “on the boundary” between clusters. If two clusters are uncovered and there is a point in the middle, then choosing this point decreases the overall cost more than any point closer to the mean of the two clusters. However, a point in the middle of two clusters and far away from their means is certainly a bad choice for a cluster center. Recently, [17] showed a better lower bound of $\Omega(\ell^3 \log^3 k / \log^2 (\ell \log k))$ while also giving an upper bound of $\mathcal{O}(\ell^3 \log^3 k)$. Still, greedy $k$-means++ is for example the default initialization method in Python’s scikit learn package [25] and our experiments confirm that this is a justified choice.

Another option is to perform improvement steps on the $k$-means++ solution before running Lloyd’s algorithm. A prominent technique in the theoretical analysis of the $k$-means problem (and other clustering problems) is to do local search by swapping in and out centers. A solution set $C$ is improved by taking out a constant number $t$ of centers and replacing them by $t$ points from $X$. [20] show that this approach with sufficiently large $t$ leads to a $(9 + \epsilon)$-approximation for the $k$-means problem. However, the approach is fairly slow since even for $t = 1$, checking if there is an improving swap takes a lot of time. Lattanzi and Sohler [21] combine local search for $t = 1$ with adaptive sampling. Instead of checking all $x \in X$ for improving swaps, they (multiple times) sample a point from $X$ by adaptive sampling and check if exchanging it for a center in $C$ improves the solution. They show that starting from an initial $k$-means++ solution, this yields a $\mathcal{O}(1)$-approximation after $\mathcal{O}(k \log \log k)$ steps. The resulting algorithm is called LS++. The analysis has since been improved by Choo et al. in [8], where it is shown that, for any $\epsilon > 0$, performing $\varepsilon k$ steps yields a $\mathcal{O}(1/\varepsilon^3)$-approximation. Figure 3 shows an idealized visualization of how improvement steps shall work: In a solution where one optimum cluster got covered twice and other clusters have been clustered together, one wants to find a point from an uncovered optimal cluster with $d^2$-sampling and then converge to an optimum solution with Lloyd’s algorithm.
Our aim in this paper is to improve the practical quality of both greedy $k$-means++ and LS++. We start with LS++, but our idea is to combine it with the strength of Lloyd’s algorithm instead of simply running local improvement steps ahead of it. We propose Foresight-LS++ (FLS++) combining steps of Lloyd’s algorithm and local search: When exchanging centers, it is not at all clear which centers should be swapped because we do not know the effect of the swap when Lloyd’s algorithm runs later. Does a particular swap really enable Lloyd’s algorithm to find good centers? The best way to answer this question is to actually perform one step of Lloyd’s algorithm for a potential swap. This way, instead of comparing the cost of the solution before and after the swap directly, we add some foresight: We perform one step of Lloyd’s algorithm and see how the solution develops depending on the swaps. The candidates for new centers are also found with $d^2$-sampling, so the centers we sample follow the same distribution. With our procedure, we may avoid putting in unfavorable centers. However, the biggest impact of our change is finding the best center to remove in a more informed way. More precisely, some of the swaps take out superfluous centers in clusters that were hit multiple times. However, there are also beneficial swaps of a different type: Here, we swap out a center that is actually the only one covering a cluster, just to better distribute the centers in that area. Figure 4 shows a schematic illustration of situations where this can be indeed helpful. By running one step of Lloyd’s algorithm, FLS++ can identify both (and potentially more) types of beneficial swaps.

Further related work. The $k$-means problem is NP-hard even for $k = 2$ [10] and $d = 2$ [23], but there exist different constant-factor approximation algorithms. The currently best approximation ratio is 6.357, achieved by Ahmadian et al. [2] with an LP rounding based approach. Prior to their work, the best known approximation ratio of $9 + \epsilon$ was achieved via a local search approach analyzed by Kanungo et al. [20]. Awasthi et al. [5] show that for general $k$ and $d$ there exists some $c > 0$ such that it is NP-hard to approximate $k$-means with factor $c$. The constant $c$ is at least 1.0013 as shown by Lee et al. [22]. The problem gets easier when bicriterial solutions are allowed, i.e., when we may select more than $k$ centers and still compare to the cost of a solution with $k$ centers. There are multiple bicriterial approximations, and Wei [27] indeed shows that for any constant $\beta > 1$, selecting $\beta k$ centers with $D^2$-sampling gives a constant factor approximation (where the constant depends on $\beta$).

There is a vast literature on Lloyd’s algorithm and various speed-up techniques. A good survey on initialization methods is due to Celebi et al. [7] who give an extensive evaluation of different heuristics to initialize Lloyd’s algorithm, including $k$-means++ and a greedy variant of it. LS++ was developed after that survey, so it is not included in the comparison. Examples for speed-up techniques are the works by Elkan [12] and Hamerly [18]. These speed up the execution of Lloyd’s algorithm but do not change the outcome.
Local Search k-means++ with Foresight

Notation. The distance of a point \( p \) to a set of centers \( C \) is \( d(p, C) = \min_{c \in C} ||p - c|| \), and for a set of points \( P \) we define the distance to a set of centers as \( d(P, C) = \sum_{p \in P} d(p, C) = \sum_{p \in P} \min_{c \in C} ||p - c|| \). We refer to the cost of \( P \) with respect to center set \( C \) as \( \Phi(P, C) = d^2(P, C) = \sum_{p \in P} d^2(p, C) = \sum_{p \in P} \min_{c \in C} ||p - c||^2 \). For an arbitrary assignment \( \phi' : P \to C \) we define \( \Phi(P, C, \phi') = \sum_{p \in P} d^2(p, \phi'(p)) \) as the cost of a clustering \( C \) using assignment \( \phi' \).

2 Algorithms

We first describe the k-means++ algorithm. It is stated in Algorithm 2 and explained below.

\begin{algorithm}[h]
\caption{d²-sampling-init.} \label{alg:sampling-init}
\begin{algorithmic}[1]
\STATE \textbf{Input:} Point set \( P \subset \mathbb{R}^d \), number \( k \in \mathbb{N} \)
\STATE \textbf{Output:} Center set \( C \subset \mathbb{R}^d \)
\STATE \hspace{1em} \( C = \emptyset \)
\FOR {\( i = 1 \) \textbf{to} \( k \)}
\STATE \hspace{2em} \( c = \text{SampleCenter}(P, C) \)
\STATE \hspace{1em} \( C = C \cup \{c\} \)
\ENDFOR
\RETURN \( C \)
\end{algorithmic}
\end{algorithm}

\begin{algorithm}[h]
\caption{k-means++.} \label{alg:k-means++}
\begin{algorithmic}[1]
\STATE \textbf{Input:} Point set \( P \subset \mathbb{R}^d \), numbers \( k, s \in \mathbb{N} \)
\STATE \textbf{Output:} Center set \( C \subset \mathbb{R}^d \)
\STATE \hspace{1em} \( C = d^2\text{-sampling-init}(P, k) \)
\STATE \hspace{1em} \( C, \phi = \text{Lloyd}(P, C, s) \)
\STATE \RETURN \( C \)
\end{algorithmic}
\end{algorithm}

The k-means++ algorithm consists of sampling \( k \) initial centers with \( d^2\)-sampling and then heuristically refining the solution by Lloyd’s method. For ease of notation in Algorithm 3, the method \( \text{Lloyd}(P, C, s) \) also outputs the optimal assignment of the points to the centers before the final centroids are computed, see the description of Lloyd’s algorithm at the beginning of the introduction. For the sampling part, we assume that we have access to a subroutine \( \text{SampleCenter}(P, C) \) which, given a point set \( P \) and center set \( C \), returns a point \( p \) from \( P \) that is sampled with probability \( d^2(p, C)/d^2(P, C) \) (unless \( C = \emptyset \), in which case the probability is \( 1/|P| \)). This is the original k-means++ sampling, also referred to as \( d^2\)-sampling. Notice that a sample can be obtained in time \( O(n) \) if all distances \( d^2(p, C) \) are already known: First, add \( d^2(p, C) \) for all \( p \in P \) to obtain the sum \( S \). Second, draw a uniform sample \( r \) from \([0, 1]\). Third, iterate through \( P \) and add up the distances again until the sum exceeds \( r = S \) for the first time; take the point before the point for which this happened. It is important that one needs all values of \( d^2(p, C) \) to do so; one can compute them in time \( O(ndk) \) when needed but it is beneficial to store values. Repeating \( \text{SampleCenter}(P, C) \) for \( k \) iterations yields the initialization part of k-means++, see Algorithm 1. We also assume that we have access to a subroutine for Lloyd’s method. Given a set \( P \) of input points, a set \( C \) of initial centers and a maximum number of steps \( s \), the method \( \text{Lloyd}(P, C, s) \) runs the loop in Step 2 of Lloyd’s algorithm as described on page 2 to compute and return a set \( C_{out} \) of centers. We define \( \Phi(\text{Lloyd}(P, C, s)) = \Phi(P, C_{out}) \). Given these subroutines, k-means++ can be described by the pseudo code in Algorithm 2.

\begin{algorithm}[h]
\caption{d²-sampling-init(P, k).} \label{alg:sampling-init}
\begin{algorithmic}[1]
\STATE \textbf{Input:} Point set \( P \subset \mathbb{R}^d \), number \( k \)
\STATE \textbf{Output:} Center set \( C \subset \mathbb{R}^d \)
\STATE \hspace{1em} \( C = \emptyset \)
\FOR {\( i = 1 \) \textbf{to} \( k \)}
\STATE \hspace{2em} \( c = \text{SampleCenter}(P, C) \)
\STATE \hspace{1em} \( C = C \cup \{c\} \)
\ENDFOR
\RETURN \( C \)
\end{algorithmic}
\end{algorithm}

The \( d^2\)-sampling-init(P, k) returns a set \( C \) of \( k \) centers with \( d^2\)-sampling. Notice that \( d^2\)-sampling can be obtained in time \( O(n) \) if all distances \( d^2(p, C) \) are already known. For the sampling part, we assume that we have access to a subroutine \( \text{SampleCenter}(P, C) \) which, given a point set \( P \) and center set \( C \), returns a point \( p \) from \( P \) that is sampled with probability \( d^2(p, C)/d^2(P, C) \) (unless \( C = \emptyset \), in which case the probability is \( 1/|P| \)). This is the original k-means++ sampling, also referred to as \( d^2\)-sampling. Notice that a sample can be obtained in time \( O(n) \) if all distances \( d^2(p, C) \) are already known: First, add \( d^2(p, C) \) for all \( p \in P \) to obtain the sum \( S \). Second, draw a uniform sample \( r \) from \([0, 1]\). Third, iterate through \( P \) and add up the distances again until the sum exceeds \( r = S \) for the first time; take the point before the point for which this happened. It is important that one needs all values of \( d^2(p, C) \) to do so; one can compute them in time \( O(nkd) \) when needed but it is beneficial to store values. Repeating \( \text{SampleCenter}(P, C) \) for \( k \) iterations yields the initialization part of k-means++, see Algorithm 1. We also assume that we have access to a subroutine for Lloyd’s method. Given a set \( P \) of input points, a set \( C \) of initial centers and a maximum number of steps \( s \), the method \( \text{Lloyd}(P, C, s) \) runs the loop in Step 2 of Lloyd’s algorithm as described on page 2 to compute and return a set \( C_{out} \) of centers. We define \( \Phi(\text{Lloyd}(P, C, s)) = \Phi(P, C_{out}) \). Given these subroutines, k-means++ can be described by the pseudo code in Algorithm 2.
Algorithm 3 Foresight-LS++.

Input: Point set \( P \subset \mathbb{R}^d \), numbers \( k, s \in \mathbb{N} \)
Parameters: \( Z \in \mathbb{N}_0 \)
Output: Center set \( C \subset \mathbb{R}^d \)

1. \( C = d^2\text{-sampling}(P,k) \)
2. \( C, \phi = \text{Lloyd}(P,C,1) \)
3. for \( i = 1 \) to \( Z \) do
   4. \( c_{\text{new}} = \text{SampleCenter}(P,C) \)
   5. \( C_{\text{min}}, \phi_{\text{min}} = \text{Lloyd}(P,C,1) \)
   6. for \( c_{\text{old}} \in C \) do
      7. \( C', \phi' = \text{Lloyd}(P,(C \setminus \{c_{\text{old}}\}) \cup \{c_{\text{new}}\},1) \)
      8. if \( \Phi(P,C',\phi') < \Phi(P,C_{\text{min}},\phi_{\text{min}}) \) then
         9. \( C_{\text{min}}, \phi_{\text{min}} = C', \phi' \)
   10. end
   11. end
12. \( C = C_{\text{min}} \)
13. end
14. \( C, \phi = \text{Lloyd}(P,C,s) \)
15. return \( C \)

The additional lines of FLS++ call Lloyd’s algorithm for one iteration, which means that all centers are replaced by the centroids of their clusters. Line 2 uses this to refine the initial sampling solution, Line 5 makes sure that we always keep centroids (instead of mixing in input points into the center set), and Line 7 is the step that gave the algorithm the prefix foresight: Instead of checking the cost of \( (C \setminus \{c_{\text{old}}\}) \cup \{c_{\text{new}}\} \), we run a Lloyd’s step to evaluate how the solution cost will develop after updating the centers. This makes it much easier for FLS++ to identify swaps that remove a superfluous center like the extra center in the upper cluster in Figure 2a.

Parameter \( Z \) is assumed to be at least \( 100000k \log \log k \) in the theoretical analysis in [21], but set to values \( \leq 25 \) in the practical evaluation [21]. After \( Z \) improvement steps, LS++ calls Lloyd’s algorithm.

LS++ does indeed improve the solution quality. But the strength in the original \( k\)-means++ algorithm lies in the fact that it uses the power of Lloyd’s to quickly do local refinements: \( d^2\)-sampling itself would not be competitive, it is the combination of finding clusters and refining the centers that makes \( k\)-means++ such a success in practical applications. Consequently, we believe that increasing the symbiosis of sampling and Lloyd’s algorithm is the key to improving LS++.

The reason is that this can be done very efficiently. Running multiple Lloyd’s steps for each \( c_{\text{old}} \) would incur a running time of \( O(ndk^2) \) for each local search step. But recomputing the centroids can be done cleverly for all points together such that one local search step in Algorithm 3 takes \( O(ndk) \) just as for LS++.
Local Search \( k \)-means++ with Foresight

Lemma 1. One iteration of the For-Loop in Lines 3-13 of Algorithm 3 can be implemented such to run in time \( O(ndk) \).

Proof. First we compute the distances of every point \( p \) to its closest center \( \phi_1(p) \) and second-closest center \( \phi_2(p) \) in time \( O(ndk) \). With this information, we can sample a point \( c_{\text{new}} \) with \( d^2 \)-sampling in time \( O(n) \). Next, we compute the distances of all points to the sample point \( c_{\text{new}} \). Then we check all \( k \) current centers whether we want to swap it out. For one candidate \( c_{\text{old}} \), we run through all \( n \) points \( p \), and determine which is its closest center. We can do this in time \( O(1) \) in the following way: if \( \phi_1(p) \neq c_{\text{new}} \), then we compare \( d(p, \phi_1(p)) \) with \( d(p, c_{\text{new}}) \). If \( \phi_1(p) = c_{\text{new}} \), then we compare \( d(p, \phi_2(p)) \) with \( d(p, c_{\text{new}}) \). After doing this for all \( c_{\text{old}} \) and all \( p \), we have determined the cluster membership of all points in time \( O(kn) \). Finally, we want to recompute the centroids and costs of all clusters which we can do in time \( O(nd) \). It is important to note that we do not recompute the reassignment of all points to the new set of centers as this would increase the overall running time \( O(nkd^2) \). This is delayed until the next iteration and thus only done for the chosen \( C_{\text{min}} \).

The approximation guarantee is also not affected.

Corollary 2. \( FLS++ \) with \( Z \geq \log n \log k \) computes an \( O(1) \)-approximation.

Proof. [21] shows that for any \( C \) with cost larger than \( 500 \cdot OPT \) and a \( c_{\text{new}} \) sampled by \( d^2 \)-sampling, there is a \( c_{\text{old}} \) such that the cost of \( C \setminus \{c_{\text{old}}\} \cup \{c_{\text{new}}\} \) compared to the cost of \( C \) is smaller than that of \( C \) by a fraction of \( 1/100k \) with probability \( 1/1000 \) (Lemma 3 in [21]). The analysis of the algorithm follows from this fact. Lloyd’s steps are always improving, since the centroid is always the best center for a cluster. Thus Lemma 3 in [21] still holds and thus does the approximation guarantee.

3 Experimental Results

Setup. The computations were performed on an Intel(R) Xeon(R) E3-1240 running at 3.7GHz and 8 cores. The code is written in C++. The code and datasets used can be found at https://github.com/lukasdrexler/flsp++/code. In the following we abbreviate \( k \)-means++ with KM++. We refer to the greedy variant of KM++ as GKM++ (analogously for (G)LS++ and (G)FLS++).

Datasets. Our experiments are based on datasets used in [21], image RGB-data used in [13], synthetically generated datasets similar to rectangles, and some datasets where the optimal solution is known for specific values of \( k \), and some new synthetically generated datasets similar to rectangles.

<table>
<thead>
<tr>
<th>dataset</th>
<th>number of points</th>
<th>dimension</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>rectangles</td>
<td>1296</td>
<td>2</td>
<td>[14]</td>
</tr>
<tr>
<td>circles</td>
<td>10000</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>close circles</td>
<td>10000</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>pr91</td>
<td>2292</td>
<td>2</td>
<td>[24]</td>
</tr>
<tr>
<td>D31</td>
<td>3100</td>
<td>2</td>
<td>[14]</td>
</tr>
<tr>
<td>s3</td>
<td>5000</td>
<td>2</td>
<td>[14]</td>
</tr>
<tr>
<td>AS</td>
<td>7500</td>
<td>2</td>
<td>[14]</td>
</tr>
<tr>
<td>Tower</td>
<td>4915200</td>
<td>3</td>
<td>[13]</td>
</tr>
<tr>
<td>Clegg</td>
<td>716320</td>
<td>3</td>
<td>[13]</td>
</tr>
<tr>
<td>Enzyme</td>
<td>1235390</td>
<td>3</td>
<td>[13]</td>
</tr>
<tr>
<td>Body measurements</td>
<td>507</td>
<td>5 (reduced)</td>
<td>[19]</td>
</tr>
<tr>
<td>Concrete strength</td>
<td>1600</td>
<td>9</td>
<td>[28]</td>
</tr>
<tr>
<td>KDD Bio Test</td>
<td>145751</td>
<td>74</td>
<td>[3]</td>
</tr>
<tr>
<td>KDD Phy Test</td>
<td>100000</td>
<td>78</td>
<td>[3]</td>
</tr>
</tbody>
</table>

For some datasets, the choice of \( k \) is obvious, as there exists a clear ground truth clustering: rectangles from Figure 2 consist of \( k = 36 \) clusters; likewise, the two generated datasets circles and close circles consist of \( k = 100 \) separated clusters as visualized in Figure 9 in the Appendix. For other datasets, we evaluated several values of \( k \): e.g. on pr91 where the optimum \( k \)-means cost is known for several values of \( k \) (see [3]).
Figure 5 Comparison on two large datasets, for $R = 50$ repetitions. GLS++ always performs 25 local search steps. For GFLS++, we display the results for performing 5, 10 and 15 such steps.

Overall performance on big data sets. To get an idea of the algorithms’ strengths and weaknesses, we compare the results of all four algorithms on three large data sets, KDD Bio Train, KDD Phy Test, and Tower. In Figure 5, we compare KM++, GKM++, GLS++ with 25 local search steps (which is the number chosen in [21]) and GFLS++ with 5, 10, and 15 steps. Since GFLS++ performs one Lloyd iteration in every local search step, 25 such steps take longer than 25 steps in GLS++. Hence, we would like to compare how GFLS++ performs when using fewer local search steps. We take a more detailed look at the trade-off between runtime and cost further below. We perform 50 runs of each configuration, i.e., each algorithm is called 50 times with a specific $k$ on each dataset. The initial centers for GLS++ and GFLS++ are chosen by greedy $d^2$-sampling. We execute GKM++, GLS++ and GFLS++ with the same initial set of centers.

As is to be expected, a larger value of $Z$ decreases runtime but increases cost. On both Phy Test and Tower, GFLS++ with $Z = 15$ obtains the overall minimum cost at the expense of a longer runtime, while outperforming GLS++ slightly at $Z = 10$ in terms of cost and runtime. Table 1 shows the maximum, average and minimum cost that the algorithms produced on each data set with $k = 100$. While the overall differences are not too big, GFLS++ consistently obtains the lowest values. Runtimes and performance comparisons for different values of $k$ and for the KDD Bio Train dataset can be seen in Section C in the appendix.

Best of repeated runs within a time limit. As we have seen in the previous section, solutions computed by GFLS++ have a smaller cost than GKM++ or GLS++ on average, but need more time to terminate. In the following we want to analyze if running KM++
Local Search k-means++ with Foresight

or LS++ multiple times can yield better results than GFLS++ in the same time. Thus, we repeat each algorithm until a time limit is reached and for each algorithm only the best solution is returned.

We choose some iteration value $B \in \mathbb{N}$ and report the best solution found by exactly $B$ repetitions of GFLS++. We then repeat (G)KM++ and GLS++ as long as their respective elapsed time is less or equal to the time used by FLS++; again, we return the best solution found.

The result of this comparison is shown in Table 2. It shows the numerical values of the final average cost of each algorithm. Additionally we evaluated the relative cost difference when compared to GFLS++. For two algorithms $A_1, A_2 \in \{\text{KM}++, \text{LS}++, \text{FLS}++\}$ and their respective average costs $c_{A_1}, c_{A_2} \in \mathbb{R}_{>0}$ we define their respective percentage cost difference as $C(A_1, A_2) := (1 - c_{A_1}/c_{A_2}) \cdot 100\%$. On data set circles the cost difference was the highest with approximately 22.37% and 8.8% when we compare the final average cost of FLS++ to KM++ and LS++. This large difference also shows that even if clusters are well separated, LS++ might still fail to find an improvement if the optimal center centroids are not close to the actual data points. In contrast, FLS++ can more often find an improvement because it evaluates the actual optimal centers for a new choice of centers, which brings the new centers closer to the actual optimal centroids. For the other datasets the relative difference in cost is not large, but a positive amount which, if we take into account the number of times each algorithms returned the smallest cost, can be achieved with large success probability.

**Performance over time.** We compare the best solutions found by all three algorithms and their average cost progression over the time. We test this procedure $R \in \mathbb{N}$ times, where the $r$-th run being defined as:

- Run GFLS++ $B$ times, let $t'_B$ be the used time until termination in run $r$.
- Repeat GKM++ and GLS++ each as long as their respective elapsed time is at most $t'_B$.

For run $r \in [R]$ let $c'_{A_r}$ for $A \in \{\text{KM}++, \text{LS}++, \text{FLS}++\}$ be the current minimum cost of algorithm $A$ at time $t \in \mathbb{R}_{\geq 0}$ and run $r$. Let $t_{\text{min}}$ be the first point in time where $A$ did terminate in every run $r \in [R]$, i.e., $c'_{A_r}$ is always defined for every time $t \geq t_{\text{min}}$. The average cost of any algorithm $A$ after $R$ runs in time step $t \geq t_{\text{min}}$ is defined as $\text{AVG}(A, t) := \frac{1}{R} \sum_{r=1}^{R} c'_{A_r}$. As before, each algorithm starts in the $r$-th run with the same set of centers from the initialization process.

| Table 1 Average cost comparison on large data sets with $k = 100$ and 50 runs. |
|---------------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| Datasets                        | KM++            | GKM++           | LS++            | FLS++ 5         | FLS++ 10        | FLS++ 15        |
| Bio Train                       | Min 1.6292E+11  | 1.6258E+11      | 1.6244E+11      | 1.6116E+11      | 1.6116E+11      | 1.6110E+11      |
|                                | Mean 1.6487E+11 | 1.6311E+11      | 1.6302E+11      | 1.6162E+11      | 1.6162E+11      | 1.6160E+11      |
|                                | Max 1.6937E+11  | 1.6579E+11      | 1.6557E+11      | 1.6240E+11      | 1.6219E+11      | 1.6219E+11      |
| Phy Test                        | Min 7.4898E+08  | 7.2525E+08      | 7.1975E+08      | 7.2427E+08      | 7.2348E+08      | 7.1737E+08      |
|                                | Mean 8.0406E+08 | 7.4972E+08      | 7.3942E+08      | 7.3905E+08      | 7.3400E+08      | 7.3128E+08      |
|                                | Max 8.5367E+08  | 7.7058E+08      | 7.5988E+08      | 7.5721E+08      | 7.5032E+08      | 7.4630E+08      |
| Tower                           | Min 1.6500E+08  | 1.6290E+08      | 1.6228E+08      | 1.6073E+08      | 1.6036E+08      | 1.6036E+08      |
|                                | Mean 1.6802E+08 | 1.6477E+08      | 1.6385E+08      | 1.6182E+08      | 1.6160E+08      | 1.6164E+08      |
|                                | Max 1.7524E+08  | 1.6755E+08      | 1.6685E+08      | 1.6365E+08      | 1.6365E+08      | 1.6365E+08      |
Table 2 Average cost. Last two columns show \((1 - c_{FLS++}/c_B) \cdot 100\%\) for \(B \in \{KM++, LS++\} \).

<table>
<thead>
<tr>
<th>data set</th>
<th>(c_{FLS++})</th>
<th>(c_{LS++})</th>
<th>(c_{KM++})</th>
<th>(C(FLS++, KM++))</th>
<th>(C(FLS++, LS++))</th>
</tr>
</thead>
<tbody>
<tr>
<td>pr91</td>
<td>9.5637E+08</td>
<td>9.5276E+08</td>
<td>9.4696E+08</td>
<td>0.98%</td>
<td>0.61%</td>
</tr>
<tr>
<td>bio train features</td>
<td>2.3918E+11</td>
<td>2.3855E+11</td>
<td>2.3864E+11</td>
<td>0.23%</td>
<td>0.09%</td>
</tr>
<tr>
<td>concrete</td>
<td>3.2724E+06</td>
<td>3.1775E+06</td>
<td>3.1308E+06</td>
<td>4.33%</td>
<td>1.47%</td>
</tr>
<tr>
<td>circles</td>
<td>3.1913E+05</td>
<td>2.7164E+05</td>
<td>2.4773E+05</td>
<td>22.37%</td>
<td>8.8%</td>
</tr>
</tbody>
</table>

Figure 6 Comparison of average cost decrease depending on the runtime of FLS++.

Figure 6 shows the progress of each algorithm over time. For LS++, we use \(Z = 25\) as number of repeat steps, as in the original experiments in [21]. We use the same value for FLS++ in the following experiments if not specified otherwise. We repeat Lloyd steps, i.e., computing memberships of all points to their closest center and recomputing the new optimal centers for this clustering, until two consecutive iterations produce solutions \(C_1, C_2\) with \(1 - \Phi(P, C_2) / \Phi(P, C_1) < 0.0001\). All plots are averaged over 100 iterations, i.e., \(R = 100\), except for (b) which is averaged over 20 iterations. In all plots, FLS++ achieves the smallest final average cost and often beats the other algorithms over the entire time frame. Similarly, LS++ is consistently better than KM++. In (c) we also compare with the known optimal solution OPT: Despite the FLS++ improvements, there is still a significant gap.

Number of best solutions found. Table 3 shows how many times KM++, LS++, or FLS++ computed the overall best solution for a given dataset within the time limit; most of the time FLS++ did return the smallest solution. Despite FLS++ having the fewest average number of iterations due to its longer running time, we can see that most of the time it
Table 3 Respective number of times each algorithm returned the smallest solution for the respective image in Figure 6 and how many iterations we could repeat the algorithm until the time limit was reached. We only count a win if the respective algorithm returned the strictly smallest cost compared to the other two algorithms in this run.

<table>
<thead>
<tr>
<th>data set</th>
<th>KM++</th>
<th>LS++</th>
<th>FLS++</th>
</tr>
</thead>
<tbody>
<tr>
<td>pr91</td>
<td>5</td>
<td>160</td>
<td>49.49</td>
</tr>
<tr>
<td>bio train features</td>
<td>87</td>
<td>50</td>
<td>8</td>
</tr>
<tr>
<td>concrete</td>
<td>0</td>
<td>357.02</td>
<td>4</td>
</tr>
<tr>
<td>circles</td>
<td>0</td>
<td>103.73</td>
<td>94</td>
</tr>
</tbody>
</table>

Varying the number of local search steps. Running LS++ and FLS++ with the same number of local search steps expectedly leads to FLS++ performing better in terms of cost, but worse in terms of runtime. But Figure 5 already hints at the possibility that using fewer local search steps in FLS++ might lead to better cost and better runtime. To this end, we fix the number of local search iterations of LS++ to 25 (guided by the experiments in [21]), while trying out different values for FLS++.

More precisely, we let the algorithms perform 50 runs in total, where one run consists of the following calls. First, we call LS++ with $Z = 25$, and afterwards FLS++ for all $Z \in \{5, \ldots, 20\}$. Importantly, we again ensure that each call in the run starts with the same set of initial centers. After completing all 50 runs, we compute the average cost and runtime of the LS++ calls across all runs, and similarly the averages for all FLS++ calls. The results are shown in Figure 8. The red line indicates the average cost (resp. time) of LS++ with $Z = 25$ over all 50 runs. Each point on the blue curve corresponds to the average cost (resp. time) of 50 calls to FLS++ with a certain number of local search iterations. We see that for FLS++ even very small values of $Z$ already suffice to beat LS++ in terms of average cost. At the same time, these small $Z$ lead to a shorter runtime, making it possible to beat LS++ both in terms of cost and time simultaneously.

To ensure that LS++ could not theoretically improve by, given some time limit, rerunning it multiple times we use a similar setup as in Section 3 to analyze the average performance for some given time bound in the Appendix, see Figure 12. In this case we use the time limit
Figure 7 Comparison of average cost decrease depending on the runtime of FLS++. Dotted lines correspond to the greedy variant of the original algorithm.

given by FLS++ for \( B = 25 \) for all algorithms. We also fixed \( Z = 25 \) for LS++ and GLS++ but varied the number of local search steps for FLS++ and GFLS++ as \( Z \in \{5, 10, 15, 20\} \). Even with fewer local search steps in FLS++ or GLS++ we get on average a smaller cost compared to LS++ or GLS++. For two considered datasets concrete and pr91, choosing \( Z = 15 \) was always enough to get approximately the same cost. For pr91, even choosing \( Z = 5 \) did result in a better solution than LS++ or GLS++ on average for the specified time limit.

4 Conclusion

We propose the new algorithm FLS++ for the \( k\)-means problem: The algorithm uses foresight by combining local search with \( d^2\)-sampling and can outperform the established methods of \( k\)-means++ and LS++ in terms of both quality and running time. Even though FLS++ only differs slightly from LS++, our experiments suggest that performing just one Lloyd iteration in each local search step often leads to better performance in terms of cost as well as runtime. Additionally, we investigate if the popularity of greedy \( d^2\)-initialization in practice is justified even though it performs poorly in theory. Surprisingly, it turns out that on most data sets considered here, all standard algorithms (as well as our new algorithm) perform substantially better when using greedy initialization.

On the other hand, running our algorithm without greedy initialization offers a robust way to recover the quality of the greedy initialization without sacrificing running time.
7:14  Local Search k-means++ with Foresight

![Graphs showing cost and time for Foresight Local Search iterations with increasing number of local search steps in FLS++ for KDD Phy Test and Tower datasets.](figures)

**Figure 8** Impact on cost and time of increasing the number of local search steps in FLS++. The dashed red line shows average cost (resp. time) of 50 runs of LS++ with 25 local search steps.

### References

Appendix

A Further datasets and cumulative costs

In the following we also show the performance of each algorithm with normal $d^2$-sampling and greedy $d^2$-sampling for some other datasets. Like in sections 3 and 3.1 we evaluate the performances of each algorithm using only normal $d^2$-sampling and then with greedy $d^2$-sampling by checking e.g. which algorithm did return the best solution in every run or the cost factors between the algorithms. Figure 10 shows the development of the average found minimum cost for each algorithm for 6 other datasets. Each dataset was averaged over 100 iterations, except for the dataset frymire, which is averaged over 20 iterations.
7:16 Local Search k-means++ with Foresight

\[ \begin{array}{c|c|c|c|c|c|c} \hline \text{data set} & \#\text{Wins KM++} & \#\text{Avg. iterations KM++} & \#\text{Wins LS++} & \#\text{Avg. iterations LS++} & \#\text{Wins FLS++} & \#\text{Avg. iterations FLS++} \\ \hline \text{close circles} & 0 & 134.27 & 0 & 37.68 & 100 & 50 \\ \hline D31 & 0 & 221.38 & 0 & 73.55 & 3 & 50 \\ \hline rectangles & 0 & 242.12 & 0 & 14.05 & 15 & 10 \\ \hline r2 & 0 & 651.87 & 0 & 88.66 & 0 & 50 \\ \hline body measurements & 0 & 403.25 & 0 & 109.13 & 100 & 50 \\ \hline \end{array} \]

We can see in Figure 10 almost the same image as for the previous cases in section 3. For datasets D31 and rectangles we additionally see that all algorithms except for KM++ almost never fail to find the optimal solution in a short amount of time.

**B Cost improvement through greedy initialization**

Now we want to consider how using \textit{greedy d}^2\text{-sampling} did improve the average costs after reaching the time limits in section 3.1.

For \( A \in \{ \text{FLS++}, \text{LS++}, \text{KM++} \} \) we define their average costs \( c_A, c_A^G \) where the superscript \( G \) indicates if we use \textit{greedy d}^2\text{-sampling}. In our case we compare both average costs over all runs \( r \in [R] \) when using time limit \( t_B^r \). Lastly, similar to the analysis above, we define the improvement factor as \((1 - \frac{c_A^G}{c_A}) \cdot 100\%\). As we can see in Table 7 for most data sets and algorithms the improvement is not below 0\% and most of the time larger than 0\%.

At last we analyse the number of times each algorithm did report the strictly best solution, the number of iterations, average costs and cost factor when compared to GFLS+++. In the case when using \textit{normal d}^2\text{-sampling} we can see in Tables 5 and 8 that GFLS+++ still on average computes the solution with the smallest cost and does so with large probability. In case of the dataset \textit{circles}, now GLS++ and GFLS+++ computed the optimal solution after the timelimit in every round. But we can see in Figure 7 that GFLS+++ does so faster than GLS+++.  

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure9.png}
\caption{Two artificially generated datasets also used in the experiments.}
\end{figure}
Table 5: Respective number of times each algorithm returned the smallest solution for the respective image in Figure 7 and how many iterations we could repeat the algorithm until the time limit was reached. We only count a win if the respective algorithm returned the strictly smallest cost compared to the other two algorithms in this run.

<table>
<thead>
<tr>
<th>data set</th>
<th>GKM++</th>
<th>GFLS++</th>
<th>GLS++</th>
<th>GFS++++</th>
<th>KM++</th>
<th>GKM++++</th>
<th>LS++</th>
<th>FLS++++</th>
<th>GFLS++++</th>
</tr>
</thead>
<tbody>
<tr>
<td>bio train features</td>
<td>5</td>
<td>104.39</td>
<td>8</td>
<td>404.61</td>
<td>87</td>
<td>10.14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>close circles</td>
<td>0</td>
<td>104.39</td>
<td>0</td>
<td>404.61</td>
<td>0</td>
<td>10.14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rectangles</td>
<td>0</td>
<td>104.39</td>
<td>0</td>
<td>404.61</td>
<td>0</td>
<td>10.14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>close circles</td>
<td>3</td>
<td>121.14</td>
<td>2</td>
<td>33.37</td>
<td>30</td>
<td>40.47</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>frymire</td>
<td>0</td>
<td>202.15</td>
<td>0</td>
<td>69.12</td>
<td>11</td>
<td>40.54</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rectangles</td>
<td>0</td>
<td>444.31</td>
<td>0</td>
<td>83.12</td>
<td>0</td>
<td>40.12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Body measurements</td>
<td>0</td>
<td>336.77</td>
<td>0</td>
<td>104.51</td>
<td>100</td>
<td>47.72</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 10: Comparison of average cost decrease depending on the runtime of FLS++. Dotted lines correspond to the greedy variant of the original algorithm.

Table 6: Respective average cost of each algorithm. Last two columns show \((1 - c_{FLS++}/c_A)\)-100% for \(A \in \{KM+++, LS++\} \).

<table>
<thead>
<tr>
<th>data set</th>
<th>GKM++</th>
<th>GFLS++</th>
<th>GLS++</th>
<th>GFS++++</th>
<th>C(FLS++, KM++)</th>
<th>C(FLS++, LS++)</th>
</tr>
</thead>
<tbody>
<tr>
<td>close circles</td>
<td>6.0468E+05</td>
<td>6.0434E+05</td>
<td>5.9341E+05</td>
<td>1.86%</td>
<td>1.81%</td>
<td></td>
</tr>
<tr>
<td>D31</td>
<td>3447.57</td>
<td>3393.26</td>
<td>3393.26</td>
<td>1.58%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>frymire</td>
<td>1.3378E+09</td>
<td>1.3287E+09</td>
<td>1.3222E+09</td>
<td>1.17%</td>
<td>0.49%</td>
<td></td>
</tr>
<tr>
<td>rectangles</td>
<td>1.5</td>
<td>1.46</td>
<td>1.46</td>
<td>2.56%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td>6.2325E+12</td>
<td>6.1887E+12</td>
<td>6.1397E+12</td>
<td>1.49%</td>
<td>0.79%</td>
<td></td>
</tr>
<tr>
<td>Body measurements</td>
<td>14804.11</td>
<td>14652.64</td>
<td>14427.42</td>
<td>3.13%</td>
<td>1.54%</td>
<td></td>
</tr>
</tbody>
</table>

Table 8: Respective average cost of each algorithm. Last two columns show \((1 - c_{GFLS++}/c_A)\)-100% for \(A \in \{GKM+++, GLS++\} \).

<table>
<thead>
<tr>
<th>data set</th>
<th>GKM++</th>
<th>GFLS++</th>
<th>GLS++</th>
<th>GFS++++</th>
<th>C(GFLS++, GKM++)</th>
<th>C(GFLS++, GLS++)</th>
</tr>
</thead>
<tbody>
<tr>
<td>p91</td>
<td>9.518E+08</td>
<td>9.518E+08</td>
<td>9.4638E+08</td>
<td>0.57%</td>
<td>0.57%</td>
<td></td>
</tr>
<tr>
<td>bio train features</td>
<td>2.390E+11</td>
<td>2.387E+11</td>
<td>2.3866E+11</td>
<td>0.15%</td>
<td>0.05%</td>
<td></td>
</tr>
<tr>
<td>concrete</td>
<td>3.1772E+06</td>
<td>3.1591E+06</td>
<td>3.1138E+06</td>
<td>2%</td>
<td>1.43%</td>
<td></td>
</tr>
<tr>
<td>circles</td>
<td>2.5481E+05</td>
<td>2.4773E+05</td>
<td>2.4773E+05</td>
<td>2.78%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>close circles</td>
<td>6.011E+05</td>
<td>6.0244E+05</td>
<td>5.9485E+05</td>
<td>1.04%</td>
<td>1.26%</td>
<td></td>
</tr>
<tr>
<td>D31</td>
<td>3393.26</td>
<td>3393.26</td>
<td>3393.26</td>
<td>0%</td>
<td>-0%</td>
<td></td>
</tr>
<tr>
<td>frymire</td>
<td>1.3321E+09</td>
<td>1.3244E+09</td>
<td>1.3244E+09</td>
<td>0.42%</td>
<td>0.58%</td>
<td></td>
</tr>
<tr>
<td>rectangles</td>
<td>1.46</td>
<td>1.46</td>
<td>1.46</td>
<td>0%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td>6.1756E+12</td>
<td>6.1742E+12</td>
<td>6.137E+12</td>
<td>0.67%</td>
<td>0.6%</td>
<td></td>
</tr>
<tr>
<td>Body measurements</td>
<td>14637.61</td>
<td>14684.68</td>
<td>14357.35</td>
<td>1.91%</td>
<td>1.56%</td>
<td></td>
</tr>
</tbody>
</table>
Table 7  Average cost decrease by using greedy $d^2$-sampling.

<table>
<thead>
<tr>
<th>data set</th>
<th>C(GKM++, KM++)</th>
<th>C(GLS++, LS+++)</th>
<th>C(GFLS++, FLS++)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pr91</td>
<td>0.48%</td>
<td>0.1%</td>
<td>0.06%</td>
</tr>
<tr>
<td>bio train features</td>
<td>0.67%</td>
<td>0.63%</td>
<td>-0.01%</td>
</tr>
<tr>
<td>concrete</td>
<td>2.91%</td>
<td>0.58%</td>
<td>0.54%</td>
</tr>
<tr>
<td>circles</td>
<td>20.16%</td>
<td>8.8%</td>
<td>0%</td>
</tr>
<tr>
<td>close circles</td>
<td>0.59%</td>
<td>0.32%</td>
<td>-0.24%</td>
</tr>
<tr>
<td>DS1</td>
<td>1.58%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Frymire</td>
<td>0.59%</td>
<td>-0.26%</td>
<td>-0.17%</td>
</tr>
<tr>
<td>rectangles</td>
<td>2.56%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>s3</td>
<td>0.87%</td>
<td>0.23%</td>
<td>0.04%</td>
</tr>
<tr>
<td>Body measurements</td>
<td>1.72%</td>
<td>0.46%</td>
<td>0.49%</td>
</tr>
</tbody>
</table>

C  Local search remaining Figures

Figure 11  Dataset concrete. Comparison of the average cost decrease for (G)LS++ and (G)FLS++ for $k = 60$, $R = 100$ and varying values of $Z$. 

(a) cost developments LS++.

(b) cost developments GLS++.

(c) cost developments FLS++.

(d) cost developments GFLS++.
Figure 12 Fixing number of local search steps for LS++ and GLS++ while using a variable number for FLS++ and GFLS++. The number after either FLS++ or GFLS++ represents the value $Z$ for this algorithm.
Figure 13 Performance boxplots for different values of $k$ and one additional dataset.
Practical Computation of Graph VC-Dimension

David Coudert
Université Côte d’Azur, Inria, I3S, CNRS, France
Mónika Csikós
IRIF, CNRS and Université Paris Cité, Paris, France
Guillaume Ducoffe
University of Bucharest, Romania
National Institute for Research and Development in Informatics, Bucharest, Romania
Laurent Viennot
Inria, DI ENS, Paris, France

Abstract
For any set system $\mathcal{H} = (V, R)$, $R \subseteq 2^V$, a subset $S \subseteq V$ is called shattered if every $S' \subseteq S$ results from the intersection of $S$ with some set in $R$. The VC-dimension of $\mathcal{H}$ is the size of a largest shattered set in $V$. In this paper, we focus on the problem of computing the VC-dimension of graphs. In particular, given a graph $G = (V, E)$, the VC-dimension of $G$ is defined as the VC-dimension of $(V, \mathcal{N})$, where $\mathcal{N}$ contains each subset of $V$ that can be obtained as the closed neighborhood of some vertex $v \in V$ in $G$. Our main contribution is an algorithm for computing the VC-dimension of any graph, whose effectiveness is shown through experiments on various types of practical graphs, including graphs with millions of vertices. A key aspect of its efficiency resides in the fact that practical graphs have small VC-dimension, up to 8 in our experiments. As a side-product, we present several new bounds relating the graph VC-dimension to other classical graph theoretical notions. We also establish the $W[1]$-hardness of the graph VC-dimension problem by extending a previous result for arbitrary set systems.

2012 ACM Subject Classification
Theory of computation → Graph algorithms analysis; Theory of computation → Parameterized complexity and exact algorithms; Theory of computation → Algorithm design techniques

Keywords and phrases VC-dimension, graph, algorithm

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.8

Supplementary Material Software (Source code): https://gitlab.inria.fr/viennot/graph-vcdim [10], archived at swh:1:dir:2edb92280298023cb1cc66ff7e5dd76b3a489b8

Funding Guillaume Ducoffe: This work was supported by a grant of the Romanian Ministry of Research, Innovation and Digitalization, CCCDI - UEFISCDI, project number PN-III-P2-2.1-PED-2021-2142, within PNCDI III. Laurent Viennot: This work was supported by the French National Research Agency (ANR) through project Tempogral with reference number ANR-22-CE48-0001.

1 Introduction
Since the seminal work of Vapnik and Chervonenkis [36], VC-dimension is one of the basic quantities describing the complexity of a set system. As such, VC-dimension is the foundation of many results in mathematics and theoretical computer science: it plays a central role in uniform sampling guarantees and is often required as an input parameter of algorithms. For instance, one of the fundamental results in computational learning theory states that a set system is PAC-learnable if and only if it has bounded VC-dimension [2]. The Sample Compression Conjecture, which has been called in [6] one of the oldest open problems in theoretical machine learning, is also related to VC-dimension [18]. VC-dimension has become
a key concept in other fields as well. In computational geometry, bounds on the VC-dimension of various geometric set systems (e.g., ones induced by half-spaces, balls or axis-parallel boxes) are essential parameters in methods developed for approximating and processing geometric data [13, 38]. Indeed, set systems with bounded VC-dimension admit structures such as small-size $\varepsilon$-nets [21] and $\varepsilon$-approximations [36, 25, 11], matchings and spanning paths of low crossing numbers [37, 7], colorings with low discrepancy [29] to name a few. We refer to the survey [28] for more details.

Practical applications of this parameter (e.g. in the design of PAC-learning algorithms), require bounds on the VC-dimension of the considered set systems. However, it was observed that the general bounds found in the literature are of limited use in practice [22]. Thus, the problem of computing the VC-dimension of set systems has attracted some attention. It is proved to be logNP-hard [32], and $W[1]$-hard for the natural parameterization by the VC-dimension [14]. Furthermore, under plausible complexity hypotheses, the VC-dimension is hard to be approximated within a sub-logarithmic factor in polynomial time [27].

There is a fast growing body of literature demonstrating the strong potential of using VC-dimension as a graph parameter. In recent studies, researchers have made significant progress in improving results in extremal and algorithmic graph theory by limiting the problems to objects with bounded VC-dimension [26, 19, 15, 17, 4, 3], including an EPTAS for the MAXIMUM CLIQUE problem [3] and subquadratic-time algorithms for diameter computation [15, 17]. The VC-dimension of a graph was also linked to the complexity of approximating a minimum-cardinality identifying code on hereditary graph classes [4].

The most commonly used notion of VC-dimension for graphs is defined as the VC-dimension of its neighbourhood set system (see Section 2 for a formal definition). More specifically, in this paper we consider the closed neighbourhoods of vertices (i.e., each vertex is included in the set of its adjacent vertices). However, we could instead consider their open neighbourhoods. Both notions result in different but comparable values for the VC-dimension. The algorithmic applications listed in [4, 15, 17] are proved using the set systems of closed neighbourhoods in a graph, while those listed in [3, 19] are proved using the set systems of open neighbourhoods. Note that the VC-dimension of other graph-related set systems has been considered in [8, 9, 17, 23], with different combinatorial and algorithmic implications. These alternative VC-dimension parameters are not considered in our paper. However, it is noteworthy that several of these parameters can be lower bounded by the graph VC-dimension. More generally, it was observed in [5] that every set system $\mathcal{H}$ can be represented as a split graph $G_{\mathcal{H}}$, in such a way that the VC-dimension of $\mathcal{H}$ is equal to the VC-dimension of the closed neighbourhoods of vertices in the stable set of $G_{\mathcal{H}}$. Therefore, graph VC-dimension is as general as the VC-dimension of arbitrary set systems, if we allow ourselves to only consider the closed neighbourhoods of a restricted subset of vertices.

Just like for general set systems, the problem of computing the VC-dimension of a graph is known to be logNP-hard [23]. However, we are not aware of any previous study on the parameterized complexity of the problem. Similarly, very few is known about the VC-dimension of complex networks. The closest such related work would be [12], where the stronger property of bounded expansion is considered. The VC-dimension of random graphs has been studied in [1], where for any fixed value $d$, a density threshold for the property of having VC-dimension at most $d$ is derived.

**Our Contributions**

While developing improved methods for graphs with bounded VC-dimension is a fruitful direction, it is just as important to provide efficient algorithms and conditions to help computing or approximating the VC-dimension of the input graph. We address this problem both from a theoretical and practical point of view.
The main contribution of this paper is a practical algorithm for computing the VC-dimension of any graph (Algorithm 1). Note that a naive algorithm for this problem would consider all vertex subsets of size at most the VC-dimension of the input. By contrast, our algorithm repeatedly updates a lower bound on the VC-dimension, so that most unexplored vertex subsets below this bound can be discarded. Furthermore, while exploring for larger shattered subsets, we use our degree-based upper bounds on the VC-dimension in order to discard at once all vertices of too small degree (at most exponential in the current lower bound). Similarly, we show that while growing a shattered subset by iteratively adding new vertices, some branches can be ignored using a simple, but surprisingly powerful, upper bound on the size of a largest shattered superset (Lemma 2). By doing so, we considerably reduce the search space, as evidenced by our experiments on some real-life networks. We implemented a few more tricks, based on a combination of bit masks and partition refinement techniques, in order to speed up some important routine tasks in the algorithm, such as: the test of whether a given subset is shattered, that of whether a search branch can be pruned, and reduction schemes for the graph to be considered. Overall, we were able to compute the VC-dimension of graphs with millions of nodes in less than 40 minutes, providing the first practical algorithm for computing graph VC-dimension. We demonstrate the efficiency of our algorithm on various practical graphs. To the best of our knowledge, this is the first analysis of the VC-dimension in real networks. Interestingly, we observe that for all graphs considered in our experiments, the VC-dimension ranges between $3$ and $8$.

Our next contribution is proving that computing the VC-dimension of a graph is a $W[1]$-hard problem for the natural parameterization by the VC-dimension (Theorem 1). For that, we revisit a previous $W[1]$-hardness proof for arbitrary set systems [14], which we combine with some insights on shattered subsets in graphs from [16].

Finally, we note that we obtain a series of new bounds on the VC-dimension with respect to classical graph parameters, such as maximum degree, degeneracy and matching number. Some of these parameters are included in the setup of [4] who show that the VC-dimension of a graph can be functionally upper bounded by any hereditary graph parameter that stays unbounded on the following graph classes: split graphs, bipartite graphs and co-bipartite graphs. However, the bounds that can be derived from [4] are rather rough, due to the use of Ramsey’s theory. By contrast, we give linear and sharp bounds for all the considered parameters.

Organization. After defining the main notions and notation of this paper in Section 2, we prove that computing the VC-dimension of graphs is $W[1]$-hard in Section 3. Then in Section 4, we summarise our bounds on the graph VC-dimension. In Section 5, we give a new exact algorithm for computing the VC-dimension of graphs and discuss several possible optimizations. In Section 6, we report our experimental results and discuss the advantages of the different optimizations methods.

2 Definitions and notation

Throughout this note, we use lowercase letters $u,v,x,y,\ldots$ for vertices, uppercase letters $X,Y,Z,\ldots$ for sets of vertices, calligraphic letters as $\mathcal{R}$ for collections of sets, and we let $\log$ denote the base 2 logarithm. Given an undirected graph $G=(V,E)$ with $|V|=n$ vertices and $|E|=m$ edges, let $N_G[v]$ denote the closed neighborhood of $v$ defined as $N_G[v]=\{u \in V \mid uv \in E \text{ or } u = v\}$. We define the degree $\deg(v)=|N_G[v]|$ of a vertex as its closed neighborhood cardinality. We use this unusual convention of counting a vertex in
its own degree for the sake of simplicity when considering closed neighborhood sizes. We also define the ball \( B_G[v, r] \) centered at a vertex \( v \) and with radius \( r \) as the set of nodes at distance at most \( r \) from \( v \). In particular, we have \( B_G[v, 1] = N_G[v] \). We omit the \( G \) subscript when \( G \) is clear from the context.

A set system \( \mathcal{H} = (V, \mathcal{R}) \) (or hypergraph) is defined by a ground set \( V \) and a collection \( \mathcal{R} \) of subsets of \( V \) called ranges. Recall that a set \( X \subseteq V \) is said to be \textit{shattered} by \( \mathcal{R} \) (or simply shattered if \( \mathcal{R} \) is clear from the context) if for every \( Y \subseteq X \) there exists a range \( R \in \mathcal{R} \) such that \( Y = R \cap X \). For any \( R \in \mathcal{R} \), the intersection \( Y = R \cap X \) is called the \textit{trace} of \( R \) on \( X \). The VC-dimension of a graph \( G \), denoted by \( \text{VCdim}(G) \), is defined as the VC-dimension of its closed neighborhood set system \( (V, \{N_G[v] \mid v \in V\}) \). A subset \( X \subseteq V \) is thus shattered if for every \( Y \subseteq X \) there exists a vertex \( v_Y \) such that its trace \( N_G[v_Y] \cap X \) on \( X \) equals \( Y \). When considering such a set system, we say that a vertex \( v \) has trace \( Y \) on a set \( X \) when its closed neighborhood has trace \( Y = N_G[v] \cap X \).

### 3 W[1]-hardness

As we have mentioned in the introduction, computing the VC-dimension of graphs is known to be \( \log \text{NP} \)-hard [23]. We show that it is also W[1]-hard for the natural parameterization by the VC-dimension by showing the following statement in Appendix A.

▶ **Theorem 1.** For any graph \( G \) and parameter \( k \leq |V(G)| \), there exists a graph \( H_G \) such that \( G \) contains a \( k \)-clique if and only if the VC-dimension of \( H_G \) is at least \( k \). Furthermore, we can construct \( H_G \) from \( G \) in \( O(k^2n^2) \) time.

### 4 Simple bounds

The following are upper bounds on the size of shattered subsets in graphs, with respect to various graph parameters. We emphasize on Lemma 2, which is a cornerstone of our practical algorithm for computing the VC-dimension of graphs (presented in the next Section 5).

▶ **Lemma 2.** Consider a shattered set \( X \) and \( Y \subseteq X \). Let \( Y' \) be the set of vertices with trace \( Y \) on \( X \). Any shattered set \( Z \) containing \( X \) satisfies \( 2^{|Z| - |X|} \leq |Y'| \).

**Proof.** For any subset \( X' \subseteq Z \setminus X \), there must exist a vertex \( v_{X'} \) with trace \( N_G[v_{X'}] \cap Z = Y \cup X' \). As \( v_{X'} \) has trace \( Y \) on \( X \), it is included in \( Y' \). All vertices \( v_{X'} \) are pairwise distinct since they have pairwise distinct traces on \( Z \). Hence, \( Y' \) has size at least \( 2^{|Z| - |X|} \). ▶

Setting \( X = Y = \{x\} \) for any vertex \( x \), we have \( Y' = N_G[x] \) and obtain the following bound.

▶ **Corollary 3.** Any shattered set \( Z \) containing a vertex \( x \) has size at most \( \lfloor \log \deg(x) \rfloor + 1 \).

In Appendix B, we additionally relate the VC-dimension of a graph with its degeneracy \( k \) and its matching number \( \nu \) (recall that a graph \( G \) is called \( k \)-degenerate if every subgraph of \( G \) contains a vertex with at most \( k \) neighbours, and that \( \nu \) is the size of a maximum matching in \( G \)). We summarize the obtained upper-bounds in the next lemma.

▶ **Lemma 4.** Let \( G \) be a non-empty graph on \( n \) vertices with maximum degree \( \Delta \), matching number \( \nu \), and degeneracy \( k \), then

\[
\text{VCdim}(G) \leq \min \{ \lfloor \log n \rfloor, \lfloor \log \Delta \rfloor + 1, k + 1, \nu + 1 \}.
\]

The following lemma allows to restrict the search of a shattered set containing a given node \( x \) to its ball \( B[x, 2] \) of radius 2.
Lemma 5. For any shattered set $X$ and $x \in X$, we have $X \subseteq B[x, 2]$.

Proof. Since $X$ is shattered, for any vertex $y \in X \setminus \{x\}$, there exists a vertex $v \in V$ such that $N[v] \cap X = \{x, y\}$. That is, $v$ is a common neighbor of $x$ and $y$ and so $\text{dist}_G(x, y) \leq 2$. □

5 Algorithm

The exact algorithm is given in Algorithm 1. Apart from the graph, it receives a lower bound $lb$ of each vertex $y$ representation $\lambda_y$ part – checking for shattered supersets – is contained in the function $\text{ExploreShattered}$ (see Algorithm 2). We encode a subset $H$ of each vertex $y$ is therefore stored in a mask $M[y]$ which is updated as we visit subsets $X$ of $H$: the $i\text{th}$ bit of $M[v]$ indicates whether the $i\text{th}$ vertex of $X$ is in $N[v]$.

Algorithm 1 $\text{VCdimComputation}(G, n, lb)$.

- **Input:** A graph $G = (V, E)$ with $n = |V|$ vertices, lower bound $lb$ on $\text{VCdim}(G)$
- **Output:** The VC-dimension of $G$

1. Let $H$ be an array containing all vertices of degree at least $2^lb$
2. Sort $H$ (optional). // We consider 3 different ways of sorting, see Section 5.3
3. Initialize a mask $M[v] \leftarrow 0$ for all $v \in H$. // Trace of $N[v]$ on $X = \emptyset$
4. Initialize $T = [T[0]] \leftarrow [n]$ // $T[y]$ counts the number of vertices $v \in V$ with trace $M[v] = y$ on $X = \emptyset$
5. For $i = 1$ to $|H|$ do
6. \[ lb \leftarrow \text{ExploreShattered}(H, i, \emptyset, T, lb) \]
7. Return $lb$

To make the algorithm more efficient, we incorporated the following key ideas:

- We fix an ordering $\prec$ of $H$ and scan subsets of $H$ in a depth first search manner.
  More precisely, $\text{ExploreShattered}$ performs a DFS of the inclusion graph of subsets of $H$ by following arcs $X \rightarrow Z$ for $X, Z \subseteq H$ such that $Z = X \cup \{z\}$ and $x \prec z$ for all $x \in X$. Starting from the empty set, any set $X = \{x_1, \ldots, x_k\}$ with $x_1 \prec \cdots \prec x_k$ is thus reachable through $\emptyset \rightarrow \{x_1\} \rightarrow \{x_1, x_2\} \rightarrow \cdots \rightarrow X$.

- For each visited set $X$, we compute a table $T$ counting for each $Y \subseteq X$ the number of vertices $v$ with trace $N[v] \cap X = Y$. If $T[Y] < 2^{lb+1-|X|}$ for some $Y$, then Lemma 2 implies that there exists no shattered set $Z \supseteq X$ of size $lb+1$ or more, so we do not explore the supersets of $X$. Note that this test is not satisfied when $X$ is not shattered as we then have $T[Y] = 0$ for some $Y \subseteq X$. The argument $X$ of $\text{ExploreShattered}$ is thus always a shattered set.

5.1 Outline of the method

In Section 5.2, we describe a way to obtain a better starting value for $lb = 0$. In Section 5.3, we describe a way to obtain a better starting value for $lb$.
Algorithm 2 \textsc{ExploreShattered}(H, i, X, T, lb).

1. Set $x \leftarrow H[i]$, $s \leftarrow |X \cup \{x\}|$, and $m \leftarrow 2^{s-1}$. // $m$ is the bit mask for $x$
2. $T' \leftarrow \text{TraceCountAdd}(T, x, m)$ // $T'[y] =$ # vertices with trace $y$ on $X \cup \{x\}$
3. $\text{prune} \leftarrow \text{False}$
4. For $y = 0$ to $2^s - 1$ do
   5. If $T'[y] < 2^{lb+1-s}$ then $\text{prune} \leftarrow \text{True}$. // by Lemma 2
6. If not $\text{prune}$ then
5. For $v \in N_G[x]$ do
   7. $M[v] \leftarrow M[v] + m$ // Update $M(v)$ to be the trace of $N[v]$ on $X \cup \{x\}$
   8. If $s > lb$ then $lb \leftarrow s$ // $X \cup \{x\}$ is shattered
10. For $j = i + 1$ to $|H|$ do
11. $lb \leftarrow \text{ExploreShattered}(H, j, X \cup \{x\}, T', lb)$
12. For $v \in N_G[x]$ do $M[v] \leftarrow M[v] - m$ // Restore the trace of $N[v]$ on $X$
13. Return $lb$

Function \text{TraceCountAdd}(T, x, m)

14. Let $T'$ be a copy of $T$ resized to $2^s$ and padded with zeros.
15. For $v \in N_G[x]$ do
17. $T'[y] \leftarrow T'[y] - 1$
18. $T'[y + m] \leftarrow T'[y] + 1$
19. Return $T'$

When considering $Z = X \cup \{x\}$, the table $T'$ for $Z$ can be obtained from $T$ in time $O(|T| + \Delta) = O(2^d + \Delta)$ where $\Delta$ is the maximum degree in $G$ and $d$ is its VC-dimension.

The overall worst case complexity of the algorithm is thus $O(n^d(2^d + \Delta))$ as we visit only shattered sets. Moreover, Corollary 3 implies $2^d = O(\Delta)$ and the complexity is thus $O(n^d\Delta)$. This pessimistic bound assumes that a constant fraction of sets of size $d$ are shattered. Empirically we observed much better running times, as reported in Section 6 with the evaluation on practical graphs. Section 5.3 describes several optimizations, including how to sort $H$.

5.2 Lower-bound computation

We compute a lower bound $lb$ by a method similar to \textsc{VCDimComputation} with $lb = 0$ (thus $H = V$). We make the search faster by only performing a partial scan of subsets of $H$. In order to find large shattered sets, we sort $H$ by non-increasing degrees. We restrict the search in two ways: we fix a maximum number $\text{maxvisits} = 64$ of times a vertex $x$ can be added to the current shattered set; we also restrict the for loop of Line 10 of \textsc{ExploreShattered} to the first $\text{maxvisits}/2$ elements. As each vertex is visited a constant number of times (at most $\text{maxvisits}$), this modified version takes linear time.

5.3 Optimizations

Vertex ordering

We have considered the following options for sorting the set $H$ of high degree vertices:
non-increasing degrees ($D^-$),
non-decreasing degrees ($D^+$),
k-core ordering ($K$): an ordering of $G_H$, the subgraph restricted to $H$, obtained by repeatedly removing a vertex with lowest degree, vertices removed first are ordered first, random ordering ($R$).
The intuition for choosing non-increasing degrees follows that of the lower bound computation: higher degree vertices tend to participate to larger shattered set and exploring them first can improve the lower bound earlier, allowing to restrict the rest of the search more severely thanks to Lemma 2. Conversely, if we start with a good enough lower bound, lower degree vertices tend to participate to smaller shattered sets and the exploration from these vertices tends to be faster. Exploring them first then speeds up the exploration from high degree vertices as we do not need to consider adding already scanned vertices anymore. Using a k-core ordering follows a similar intuition with the refinement of taking into account the degree after removing previous vertices rather than the degree in the full graph. Using a random ordering seems a basic choice for comparison.

Ball restriction
Lemma 5 implies that when starting an exploration from $x = H[i]$ (Line 6 of VCDIMCOMPUTATION) we can restrict the search to consider only vertices in $B[x, 2] \cap \{H[i], H[i + 1], \ldots\}$.

Graph reduction
As we focus on shattered sets included in $H$, we can restrict the graph while preserving all possible traces on $H$. For that purpose, for each possible trace $Y \subseteq H$ which can be obtained as $H \cap N[v]$ for $v \in V$, we keep at most one vertex $v \in V \setminus H$ with trace $N[v] \cap H = Y$. Such a selection can be obtained in linear time using partition refinement [31, 20] as follows. Starting from the partition $P = \{V\}$ we iteratively refine it by sets $N[x]$ for $x \in H$: each refinement step consists in splitting each part $P \in P$ into $P \cap N[X]$ and $P \setminus N[X]$ (if one of the two sets is empty, $P$ remains unchanged). Each refinement step clearly maintains the invariant that all vertices in a part have the same trace on the set of vertices of $H$ processed so far. At the end of the process, all vertices in a part must have the same trace on $H$. We thus keep one vertex per final part not intersecting $H$ and all vertices in $H$ to obtain a set $V' \supseteq H$ of vertices providing the same traces on $H$ as $V$ and proceed on $G_{V'}$ instead of $G$.

6 Experiments
6.1 Dataset
We evaluate the performance of our algorithm on various types of practical graphs. We use graphs from the BioGRID interaction database (BIO-*) [30]; a protein interactions network (dip20170205) [33]; and graphs of the autonomous systems from the Internet (oregon2, CAIDA_as and DIMES) [24, 35, 34]. We also test computer networks (Gnutella, Skitter), web graphs (notreDame and BerkStan), social networks (Epinions, Facebook, Twitter), co-author graphs (ca-HepPh, dblp), road networks (t.CAL, t.FLA), a 3D triangular mesh (buddha), a graph from a computer game (FrozenSea), and grid-like graphs from VLSI applications (z-alue7065). The data is available from snap.stanford.edu, webgraph.di.unimi.it, www.dis.uniroma1.it/challenge9, graphics.stanford.edu, steinlib.zib.de, and movingai.com. Furthermore, we use synthetic inputs: grid300-10 and grid500-10
Table 1 The graphs we use with their main parameters and the time (in seconds) required by our reference implementation KBG to compute their VC-dimension.

<table>
<thead>
<tr>
<th>Graph</th>
<th>#nodes</th>
<th>#edges</th>
<th>max.deg.</th>
<th>VC-dim</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-MV-Physical-3.5</td>
<td>21866</td>
<td>68123</td>
<td>1117</td>
<td>5</td>
<td>0.11</td>
</tr>
<tr>
<td>BIO-SYS-Aff-Cap-MS-3.5</td>
<td>38926</td>
<td>299855</td>
<td>2193</td>
<td>7</td>
<td>4.41</td>
</tr>
<tr>
<td>BIO-SYS-Aff-Cap-RNA-3.5</td>
<td>12716</td>
<td>40541</td>
<td>3571</td>
<td>7</td>
<td>0.03</td>
</tr>
<tr>
<td>dip20170205</td>
<td>27029</td>
<td>73762</td>
<td>289</td>
<td>5</td>
<td>0.14</td>
</tr>
<tr>
<td>oregon2-010331</td>
<td>10900</td>
<td>31180</td>
<td>2343</td>
<td>6</td>
<td>0.20</td>
</tr>
<tr>
<td>CAIDA-as-20130601</td>
<td>44611</td>
<td>151434</td>
<td>3962</td>
<td>7</td>
<td>11.50</td>
</tr>
<tr>
<td>DIMES-201204</td>
<td>25367</td>
<td>75004</td>
<td>3781</td>
<td>7</td>
<td>0.16</td>
</tr>
<tr>
<td>as-skitter</td>
<td>1696415</td>
<td>11095298</td>
<td></td>
<td>8</td>
<td>2116.90</td>
</tr>
<tr>
<td>p2p-Gnutella09</td>
<td>8114</td>
<td>26013</td>
<td>102</td>
<td>5</td>
<td>0.01</td>
</tr>
<tr>
<td>gnutella31</td>
<td>62586</td>
<td>147892</td>
<td>95</td>
<td>4</td>
<td>0.29</td>
</tr>
<tr>
<td>notreDame</td>
<td>325729</td>
<td>1090108</td>
<td>10721</td>
<td>6</td>
<td>2.94</td>
</tr>
<tr>
<td>y-BerkStan</td>
<td>685231</td>
<td>6649470</td>
<td>538280</td>
<td>7</td>
<td>11.44</td>
</tr>
<tr>
<td>ca-HepPh</td>
<td>12008</td>
<td>118489</td>
<td>491</td>
<td>5</td>
<td>46.53</td>
</tr>
<tr>
<td>com-dblp</td>
<td>317080</td>
<td>1049866</td>
<td>343</td>
<td>5</td>
<td>2.30</td>
</tr>
<tr>
<td>epinions1</td>
<td>75888</td>
<td>405740</td>
<td>3044</td>
<td>7</td>
<td>58.69</td>
</tr>
<tr>
<td>facebook-combined</td>
<td>4039</td>
<td>88234</td>
<td>1045</td>
<td>6</td>
<td>8.15</td>
</tr>
<tr>
<td>twitter-combined</td>
<td>81306</td>
<td>1342296</td>
<td>3383</td>
<td>7</td>
<td>456.22</td>
</tr>
<tr>
<td>t.CAL</td>
<td>1890816</td>
<td>2315222</td>
<td>7</td>
<td>3</td>
<td>4.56</td>
</tr>
<tr>
<td>t.FLA</td>
<td>1070377</td>
<td>1343951</td>
<td>8</td>
<td>3</td>
<td>2.53</td>
</tr>
<tr>
<td>buddha</td>
<td>543652</td>
<td>1631574</td>
<td>17</td>
<td>4</td>
<td>3.21</td>
</tr>
<tr>
<td>froz</td>
<td>754197</td>
<td>2902525</td>
<td>8</td>
<td>3</td>
<td>4.70</td>
</tr>
<tr>
<td>z-alue7065</td>
<td>34047</td>
<td>54841</td>
<td>4</td>
<td>3</td>
<td>0.03</td>
</tr>
<tr>
<td>grid300-10</td>
<td>90601</td>
<td>162353</td>
<td>4</td>
<td>3</td>
<td>0.21</td>
</tr>
<tr>
<td>xgrid500-10</td>
<td>251001</td>
<td>450766</td>
<td>4</td>
<td>3</td>
<td>0.61</td>
</tr>
<tr>
<td>powerlaw2.5</td>
<td>1000000</td>
<td>1916356</td>
<td>9447</td>
<td>6</td>
<td>3.79</td>
</tr>
</tbody>
</table>

are square grids with respective sizes 301 × 301, and 501 × 501 where 10% of the edges were randomly deleted; and powerlaw2.5 is a random graph generated according to the configuration model with a degree sequence following a power law with exponent 2.5.

All experiments were performed on a cluster of 20 nodes equipped with two Cascade Lake Intel Xeon 5216 16 cores processors at 2.4GHz and 192GB of memory each. Sixteen processes were run in parallel on four nodes of the cluster. Our computation times are thus pessimistic compared to running each process on a fully reserved node (for example, the computation of the VC-dimension of the twitter graph of our dataset takes 431s on a fully reserved node compared to 456s in our experiment even though we report user times). The code is available at gitlab.inria.fr/viennot/graph-vcdim.

6.2 Graphs and reference time

We first present our dataset graphs in Table 1 together with their VC-dimension and time required to compute it with our reference implementation KBG: which uses k-core ordering (K), ball restriction (B) and graph reduction (G). We have chosen KBG as our reference choice of optimizations as it provides the minimum sum of running times over all graphs of our dataset. We list also their size in terms of number of nodes (not counting isolated nodes) and number of edges (duplicate edges are removed).
We observe that the VC-dimension of all graphs in the dataset is rather small: at most 8, even for graphs with millions of nodes and over 10M edges (M stands for million). Moreover, its computation with our KBG implementation takes at most a few seconds for most of the graphs, and less than a few minutes for all but one: as-skitter for which it takes around 35 minutes. Memory usage (not reported here) grows with the input graph size up to roughly 600 megabytes for as-skitter. Not surprisingly, this most difficult graph is both the largest in terms of number of edges and the most complex in terms of VC-dimension. We analyze the dependency of the computation time with respect to some graph parameters in Section 6.3.

To appreciate these running times, recall that our algorithm for computing the VC-dimension consists in first computing a lower-bound $lb$, then identifying the set $H$ of high degree nodes, that is those with degree at least $2^{lb}$, and then ordering this set for exploring shattered sets included in $H$. Figure 1 shows that running times are basically proportional to the number $x$ of shattered sets considered by KBG, which is estimated as follows. For each visited shattered set $X = \{v_1, \ldots, v_k\}$ with $v_1 \prec \cdots \prec v_k$, our algorithm tries to add high degree nodes of $B[v_1, 2]$ coming after $v_k$ in the ordering $\prec$ used for $H$. We can thus estimate $x$ as the product of the number $s$ of visited shattered sets multiplied by half of the average ball size $bsize = \frac{1}{|H|} \sum_{v \in H} \|B[v, 2] \cap H\|$. These numbers are reported in Table 2. We see in Figure 1 that most networks are close to the black line that corresponds to a rate of 2 millions tentative shattered sets per second. For low values of $x$ five networks appear significantly above the line: p2p-gnutella09, BIO-SYS-Aff-Cap-RNA-3.5, DIMES-201204, powerlaw2.5 and buddha (from left to right). This is due to the overhead of reading the input file, computing the lower-bound, the $k$-core ordering, and the graph reduction which appears to be higher than the time for exploring shattered sets in these networks (we get comparable times when running all these phases and stopping before exploring).
### Table 2 VC-dimension \( d \), counts of shattered sets and high degree nodes, average ball size.

<table>
<thead>
<tr>
<th>Graph</th>
<th>( d )</th>
<th>all ( \deg \geq 2^d )</th>
<th>Lem. 2</th>
<th>Lem. 2G</th>
<th>#deg ( \geq 2^d )</th>
<th>KBG</th>
<th>#deg ( \geq 2^d )= bsize</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-MV-Physical-3.5</td>
<td>5</td>
<td>6 990 780</td>
<td>1 619 098</td>
<td>4 474</td>
<td>4 242</td>
<td>582</td>
<td>1 242</td>
</tr>
<tr>
<td>BIO-SYS-Aff-Cap-MS-3.5</td>
<td>7</td>
<td>–</td>
<td>419 882 850</td>
<td>133 927</td>
<td>133 546</td>
<td>455</td>
<td>133 546</td>
</tr>
<tr>
<td>BIO-SYS-Aff-Cap-RNA-3.5</td>
<td>7</td>
<td>1 552 654 200</td>
<td>9 438</td>
<td>1 388</td>
<td>1 246</td>
<td>19</td>
<td>1 292</td>
</tr>
<tr>
<td>dip20170205</td>
<td>5</td>
<td>9 050 178</td>
<td>2 823 068</td>
<td>14 503</td>
<td>14 227</td>
<td>586</td>
<td>14 227</td>
</tr>
<tr>
<td>oregon2-010331</td>
<td>6</td>
<td>18 763 142</td>
<td>1 777 582</td>
<td>2 382</td>
<td>2 610</td>
<td>105</td>
<td>2 715</td>
</tr>
<tr>
<td>Caida-Aas-20130601</td>
<td>7</td>
<td>–</td>
<td>134 557 021</td>
<td>179 232</td>
<td>174 320</td>
<td>168</td>
<td>441 741</td>
</tr>
<tr>
<td>DIMES-201204</td>
<td>7</td>
<td>345 796 843</td>
<td>1 982 414</td>
<td>4 206</td>
<td>3 870</td>
<td>67</td>
<td>3 870</td>
</tr>
<tr>
<td>as-skitter</td>
<td>8</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>g2p-Gneatellao00</td>
<td>5</td>
<td>1 307 927</td>
<td>399 096</td>
<td>433</td>
<td>420</td>
<td>58</td>
<td>420</td>
</tr>
<tr>
<td>gnutella31</td>
<td>4</td>
<td>1 781 293</td>
<td>131 652</td>
<td>3 958</td>
<td>2 135</td>
<td>1 938</td>
<td>19 417</td>
</tr>
<tr>
<td>noteDane</td>
<td>6</td>
<td>294 870 046</td>
<td>46 413 353</td>
<td>5 022</td>
<td>2 983</td>
<td>2 963</td>
<td>26 733</td>
</tr>
<tr>
<td>y-BerkStan</td>
<td>7</td>
<td>–</td>
<td>33 179 823</td>
<td>108 641</td>
<td>90 394</td>
<td>1 540</td>
<td>90 394</td>
</tr>
<tr>
<td>ca-HepPh</td>
<td>5</td>
<td>1 50 848 664</td>
<td>137 719 903</td>
<td>1 046 942</td>
<td>1 037 751</td>
<td>1 529</td>
<td>1 037 751</td>
</tr>
<tr>
<td>com-dblp</td>
<td>5</td>
<td>17 874 027</td>
<td>2 489 073</td>
<td>24 748</td>
<td>14 329</td>
<td>4 280</td>
<td>65 024</td>
</tr>
<tr>
<td>epinions1</td>
<td>7</td>
<td>–</td>
<td>–</td>
<td>1 028 033</td>
<td>1 025 863</td>
<td>1 162</td>
<td>1 025 963</td>
</tr>
<tr>
<td>facebook-combined</td>
<td>7</td>
<td>6 722 73 307</td>
<td>568 585 300</td>
<td>389 816</td>
<td>389 471</td>
<td>882</td>
<td>389 471</td>
</tr>
<tr>
<td>twitter-combined</td>
<td>7</td>
<td>–</td>
<td>–</td>
<td>1 709 219</td>
<td>1 707 213</td>
<td>2 283</td>
<td>2 562 770</td>
</tr>
<tr>
<td>uCAl</td>
<td>3</td>
<td>8 107 636</td>
<td>12</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>t.FLA</td>
<td>3</td>
<td>4 805 712</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>buddha</td>
<td>4</td>
<td>7 928 071</td>
<td>21</td>
<td>17</td>
<td>0</td>
<td>0</td>
<td>270 936</td>
</tr>
<tr>
<td>fox</td>
<td>3</td>
<td>17 402 814</td>
<td>16 370 680</td>
<td>2 033 860</td>
<td>2 026 837</td>
<td>691 850</td>
<td>2 026 837</td>
</tr>
<tr>
<td>x-alb17065</td>
<td>3</td>
<td>250 178</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>grid300-10</td>
<td>3</td>
<td>800 788</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>xgrid500-10</td>
<td>3</td>
<td>2 222 193</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>powerlaw2.5</td>
<td>6</td>
<td>236 284 923</td>
<td>31 023 430</td>
<td>4 653</td>
<td>2 674</td>
<td>886</td>
<td>2 674</td>
</tr>
</tbody>
</table>

### 6.3 Analysis: shattered sets and high degree nodes

The running time of our computation is mostly governed by the number of high degree nodes and the number of shattered sets explored, we thus present a detailed analysis of them. Table 2 lists several measures we could perform as follows. By removing the pruning according to Lemma 2, our algorithm explores all shattered sets. By setting an initial lower-bound of 0, we first tried to compute all shattered sets and report their number in column “all”. Note that this computation was not doable within our timeout of 6 hours for six graphs. Unsurprisingly, bigger values are observed for larger VC-dimension. We then tried to compute all shattered sets included in the set \( H' \) of nodes with degree at least \( 2^d \) where \( d \) is the VC-dimension of the graph. Again this computation was out of reach within our time limit for three graphs. We then use KB with \( d \) as lower-bound to obtain the number of visited shattered sets in \( H' \) according to Lemma 2 in column “Lem.2”. We obtain similarly column “Lem.2G” using KBG. Column “KBG” is obtained using KBG with its heuristic lower-bound (instead of the exact value \( d \)). We also report the size of \( H' \) and \( H \) (columns “\#deg \( \geq 2^d \)” and “\#deg \( \geq 2^d \)= bsize” respectively), and average ball size bsize in the last column.

We observe that restricting to \( H' \) can reduce the number of shattered sets by a huge factor (e.g. BIO-SYS-Aff-Cap-RNA-3.5). Note that \( H' \) can be empty in graphs with low maximum degree such as grids, explaining the zeros in the table. In general, Lemma 2 allows to further reduce the number \( y \) of shattered sets to explore as illustrated by Figure 2 (column “\#deg \( \geq 2^d \)” versus “Lem.2” of Table 2). For networks with more than 10k shattered sets in \( H' \), we observe a reduction factor varying from roughly 10 for networks with VC-dimension 3-4, to roughly 100 for networks with VC-dimension 5, and from 1000 to 10k for networks with VC-dimension 6-7 except for BIO-SYS-Aff-Cap-RNA-3.5 for which the number \( x \) of shattered sets in \( H' \) was already quite low. We could not compute the value \( x \) for our only network of VC-dimension 8 within our 6 hours limit. Overall, this shows the efficiency of our approach by restricting to high degree nodes and pruning the search by Lemma 2.
Figure 2: The number $y$ of visited shattered sets versus the number $x$ of shattered sets in $H'$: each network in the dataset is represented by a disk with coordinates $(x, y)$, whose color indicates the VC-dimension $d$ of the network, while its size is proportional to the logarithm of $|H'|$ ($H'$ denotes the set of nodes with degree $2^d$ at least).

6.4 Lower and upper bounds

As detailed in Appendix C.1, the lower bounds we obtain with our heuristic are mostly equal to the true VC-dimension or just one less. The upper bounds presented in Lemma 4, except for the matching number, are analyzed in Appendix C.2 and are often much larger than the true VC-dimension except for grid-like graphs.

6.5 Optimizations

We now compare our reference implementation with other variants of our algorithm obtained by changing the ordering of the vertices ($D^-$, $D^+$, $K$, $R$), using ball restriction ($B$) or not, and using graph restriction ($G$) or not (see Table 3).

Concerning the ordering of the nodes used for scanning shattered sets, we first note that non-decreasing degrees ($D^+$) is almost always faster than non-increasing degrees ($D^-$). Notable exceptions are notreDame and twitter-combined where our initial lower bounds are $\text{VCdim}(\text{notreDame}) - 2$ and $\text{VCdim}(\text{twitter-combined}) - 1$ respectively. By observing our traces of execution, we explain it by the fact that the non-increasing order allows to find faster a better lower-bound, which then speeds up the rest of the computation. When the starting lower-bound was indeed exact, non-decreasing degrees is always faster or at least very close to non-increasing degrees. Our intuition is that the number of tentative shattered sets inspected is lower in that case. Indeed, for shattered sets constructed from the first nodes of the ordering, we have to consider all possible remaining nodes (that are in their ball of radius two) and try to construct a tentative shattered set by adding each of them. Putting
Table 3 Comparing different optimization options of our algorithm with our reference selection (KBG). A dash (--) indicates that the timeout of 6 hours was reached.

<table>
<thead>
<tr>
<th>Graph</th>
<th>D^-BG</th>
<th>D^+BG</th>
<th>KBG</th>
<th>KBG</th>
<th>KG</th>
<th>KB</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-MV-Physical-3.5</td>
<td>0.22</td>
<td>0.09</td>
<td>0.11</td>
<td>0.22</td>
<td>0.17</td>
<td>0.13</td>
<td>0.19</td>
</tr>
<tr>
<td>BIO-SYS-Aff-Cap-MS-3.5</td>
<td>18.09</td>
<td>3.06</td>
<td>4.41</td>
<td>5.69</td>
<td>6.50</td>
<td>4.58</td>
<td>6.86</td>
</tr>
<tr>
<td>BIO-SYS-Aff-Cap-RNA-3.5</td>
<td>0.03</td>
<td>0.01</td>
<td>0.03</td>
<td>0.06</td>
<td>0.04</td>
<td>0.04</td>
<td>0.06</td>
</tr>
<tr>
<td>dip20170205</td>
<td>0.31</td>
<td>0.12</td>
<td>0.14</td>
<td>0.18</td>
<td>0.31</td>
<td>0.17</td>
<td>0.35</td>
</tr>
<tr>
<td>oregon2-010331</td>
<td>0.54</td>
<td>0.55</td>
<td>0.20</td>
<td>0.29</td>
<td>0.20</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td>CAIDA-as-20130601</td>
<td>11.76</td>
<td>17.26</td>
<td>11.50</td>
<td>6.76</td>
<td>11.85</td>
<td>14.10</td>
<td>14.06</td>
</tr>
<tr>
<td>DIMIES-201204</td>
<td>0.15</td>
<td>0.08</td>
<td>0.16</td>
<td>0.11</td>
<td>0.09</td>
<td>0.13</td>
<td>0.13</td>
</tr>
<tr>
<td>as-skitter</td>
<td>13830.13</td>
<td>3673.65</td>
<td>2116.90</td>
<td>4022.75</td>
<td>8892.87</td>
<td>2358.56</td>
<td>10102.42</td>
</tr>
<tr>
<td>p2p-Gnutella09</td>
<td>0.02</td>
<td>0.01</td>
<td>0.01</td>
<td>0.03</td>
<td>0.02</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>gnutella31</td>
<td>0.17</td>
<td>0.30</td>
<td>0.29</td>
<td>0.32</td>
<td>15.20</td>
<td>4.41</td>
<td>16.77</td>
</tr>
<tr>
<td>notreDame</td>
<td>1.87</td>
<td>14.36</td>
<td>2.94</td>
<td>2.85</td>
<td>55.95</td>
<td>4.41</td>
<td>122.59</td>
</tr>
<tr>
<td>y-BerkStan</td>
<td>11.83</td>
<td>11.32</td>
<td>11.44</td>
<td>8.33</td>
<td>56.86</td>
<td>38.47</td>
<td>521.76</td>
</tr>
<tr>
<td>ca-HepPh</td>
<td>138.93</td>
<td>33.54</td>
<td>46.53</td>
<td>61.00</td>
<td>47.03</td>
<td>47.82</td>
<td>48.18</td>
</tr>
<tr>
<td>com-dblp</td>
<td>1.79</td>
<td>2.23</td>
<td>2.30</td>
<td>1.71</td>
<td>82.82</td>
<td>2.68</td>
<td>170.05</td>
</tr>
<tr>
<td>opinions1</td>
<td>511.94</td>
<td>31.08</td>
<td>58.69</td>
<td>144.99</td>
<td>58.69</td>
<td>63.67</td>
<td>63.07</td>
</tr>
<tr>
<td>facebook-combined</td>
<td>21.37</td>
<td>10.30</td>
<td>8.15</td>
<td>12.09</td>
<td>44.56</td>
<td>8.24</td>
<td>45.04</td>
</tr>
<tr>
<td>twitter-combined</td>
<td>1079.24</td>
<td>3132.27</td>
<td>456.22</td>
<td>397.94</td>
<td>1532.12</td>
<td>473.12</td>
<td>1549.07</td>
</tr>
<tr>
<td>t.CAL</td>
<td>4.59</td>
<td>4.41</td>
<td>4.56</td>
<td>4.45</td>
<td>4.56</td>
<td>4.85</td>
<td>4.87</td>
</tr>
<tr>
<td>t.FLA</td>
<td>2.42</td>
<td>2.35</td>
<td>2.53</td>
<td>2.47</td>
<td>2.46</td>
<td>2.62</td>
<td>2.58</td>
</tr>
<tr>
<td>buddha</td>
<td>2.99</td>
<td>2.92</td>
<td>3.21</td>
<td>3.01</td>
<td>2606.40</td>
<td>3.22</td>
<td>4525.09</td>
</tr>
<tr>
<td>froz</td>
<td>4.39</td>
<td>4.40</td>
<td>4.70</td>
<td>6.44</td>
<td>–</td>
<td>4.45</td>
<td>–</td>
</tr>
<tr>
<td>z-value7065</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
<td>0.04</td>
<td>0.03</td>
<td>0.04</td>
<td>0.03</td>
</tr>
<tr>
<td>grid300-10</td>
<td>0.22</td>
<td>0.23</td>
<td>0.21</td>
<td>0.21</td>
<td>0.21</td>
<td>0.22</td>
<td>0.21</td>
</tr>
<tr>
<td>xgrid500-10</td>
<td>0.61</td>
<td>0.59</td>
<td>0.61</td>
<td>0.63</td>
<td>0.62</td>
<td>0.62</td>
<td>0.65</td>
</tr>
<tr>
<td>powerlaw2.5</td>
<td>4.06</td>
<td>3.59</td>
<td>3.79</td>
<td>3.77</td>
<td>3.66</td>
<td>7.88</td>
<td>8.90</td>
</tr>
</tbody>
</table>

Nodes with lower degree first seem to result in a better balance with respect to the number of tentative shattered sets tested for being shattered. In that respect, the k-core ordering (K) seems to work slightly better since it considers the remaining degrees after removing the first nodes rather than degree in the full graph. We note however that a random ordering (R) gives overall good results and RBG can even outperform KBG when our lower-bound is not exact, similarly as D^-BG can outperform D^+BG. This is in particular the case for twitter-combined. Overall there is no clearly better strategy for the ordering and both k-core ordering and random ordering seem legitimate choices.

Recall that the ball optimization (B) consists in restricting the nodes added to a shattered set X to those that are in the ball of radius two centered at the first node of X. It also allows to reduce the number of tentative shattered sets tested and is almost always beneficial. It appears to be mandatory on graphs with very large number of high degree nodes such as froz (the number of high degree nodes is analyzed in Section 6.3).

The graph reduction optimization (G) is not very costly in terms of computation time (KBG is always almost as good as KB) and gives a significant improvement on difficult graphs such as as-skitter and y-BerkStan.

6.6 VC-dimension of random graphs

We computed the average VC-dimension of various Erdős-Rényi random graphs \( G_{n,p} \) (where each edge appears independently with probability \( p \)) with up to \( n = 400 \) nodes and compared them to [1] which proved a threshold of \( p = n^{-11/20} = n^{-0.55} \) above which the VC-dimension
Figure 3. Top: the average VC-dimension of $G_{n,p}$ as a function of $p$ for $n = 32, 45, 64, 100, 128$. Bottom: a zoom on values $p \in [0, 0.3]$ including additional curves for $n = 256, 400$.

$d$ of $G_{n,p}$ tends to be greater than 3, and similarly $p = n^{-21/55} \approx n^{-0.38}$ for $d > 4$, and $p = n^{-7/24} \approx n^{-0.29}$ for $d > 5$. On the one hand, our results confirm the $p = n^{-0.55}$ and $p = n^{-0.38}$ thresholds which appear to be rather sharp already for $n = 100$ or $n = 256$. On the other hand, observing the $p = n^{-0.29}$ threshold seems to require size $n$ greater than 400 (see Figure 3).

Figure 4 shows the VC-dimension of various power-law random graphs for $n \in \{32, 64, 128, 256, 512, 1000, 2000, 4000, 30000, 300000, 1000000\}$ nodes and exponent $\beta \in [1, 4]$. A graph with $n$ nodes and exponent $\beta$ is obtained by generating a sequence of degrees following a power-law with parameter $\beta$ (deg($u$) gets value $d$ with probability proportional to $1/d^\beta$) and generating a graph according to the configuration model (we generate deg($u$) half edges for each node $u$ and connect half edges according to a random permutation). For each pair $n, \beta$, ...
we have generated 20 random graphs and computed the average VC-dimension of them. Our curves seem to indicate a threshold value near $\beta = 3$ below which the VC-dimension tends to infinity as $n$ grows while the VC-dimension seems to remain constant above (at most three for large $n$).

7 Conclusion

A first conclusion is that there is room for improvement with respect to the quick estimation of the VC-dimension, especially concerning good upper bounds. Our work can be directly applied in the setups, where the hyperedges are induced by the open neighborhoods or where only the neighborhoods of a subset of vertices are considered as hyperedges. In future works, besides studying the VC-dimension of power-law random graphs as mentioned above, we will consider extending our algorithm to the practical computation of other VC-dimension parameters on graphs (e.g., the distance VC-dimension, see [8]), and of the VC-dimension of arbitrary set systems where the ranges are given explicitly.

References


A Proof of Theorem 1

Proof. Our starting point is the FPT-reduction of Downey et al. [14] who showed that computing the VC-dimension of general set systems is W[1]-hard by the following reduction from $k$-clique detection.

\begin{theorem} [Theorem 3 in [14, Sec. 5]] \label{thm:VC-dimension}
Let $G = (V, E)$ be a graph and $k \leq |V|$ be a parameter. Then $G$ contains a $k$-clique if and only if there is a shattered $k$-subset in the set system $\mathcal{H}_G = (X, R)$, where $X = V \times \{1, 2, \ldots, k\}$ and $R = R_0 \cup R_1 \cup R_2 \cup R_3$ with
\end{theorem}
We construct a graph $Y$. Assume that each of the $v$, $y$, some neighbor $x$, $z$ satisfies

$\mathcal{R}_1 = \{(v, i) \mid v \in V, 1 \leq i \leq k\}$;

$\mathcal{R}_2 = \{(u, i), (v, j) \mid uv \in E, 1 \leq i, j \leq k\}$;

$\mathcal{R}_3 = \{(v, i) \mid v \in V, i \in S\} \mid S \subseteq \{1, 2, \ldots, k\}$ and $|S| \geq 3$.

Let $H_G = (X, \mathcal{R})$ be defined as in Theorem 6. Note that the cumulative cardinalities of all sets in $\mathcal{R}$ sum up to $O(kn + k^2m + k^2n) = O(k^2n^2)$.

We construct a graph $H_G$ from $H_G$ as follows:

- $V(H_G) = X \cup \mathcal{R}$;
- for every $x \in X$ and $R \in \mathcal{R}$ such that $x \in R$, add the edge $\{x, R\}$ to $E(H_G)$;
- if $k \geq 3$, for every distinct $x, y \in X$ add the edge $\{x, y\}$ to $E(H_G)$.

Note that the cost of computing $H_G$ is that of computing $H_G$, with an additional cost in $O(|X|^2) = O(k^2n^2)$ if $k \geq 3$. It is easy to see that if $Y \subseteq X$ is shattered in $H_G$, then it is also shattered by the closed neighborhoods in $H_G$. Thus, Theorem 6 implies that if $G$ has a $k$-clique, then the VC-dimension of $H_G$, and so the VC-dimension of $H_G$, is at least $k$.

It remains to show that if $G$ has no $k$-clique (or equivalently, the VC-dimension of $H_G$ is strictly less than $k$), then the VC-dimension of $H_G$ is at most $k - 1$.

If $k = 0$, then $G$ must be an empty graph. Then, $\mathcal{R} = \mathcal{R}_0$. It implies that $H_G$ is reduced to one vertex, and so, its VC-dimension equals 0.

If $k = 1$, then $G$ is a stable set which implies $\mathcal{R}_2 = \emptyset$, and we also have $\mathcal{R}_3 = \emptyset$. In particular, $\mathcal{R} = \mathcal{R}_0 \cup \mathcal{R}_1$. By definition, $H_G$ contains a perfect matching between $X$ and $\mathcal{R}_1$, plus an isolated vertex for $\mathcal{R}_0$. Therefore, the VC-dimension of $H_G$ equals 1.

Assume now that $\omega(G) \leq k - 1$ with $k \geq 3$ and suppose for the sake of contradiction that there exists a $k$-element subset $Y \subseteq V(H_G)$ which is shattered by closed neighborhoods in $H_G$. By definition, $H_G$ is a split graph with clique $X$ and independent set $\mathcal{R}$.

> Claim. We either have $Y \subseteq X$ or $Y \subseteq \mathcal{R}$.

Proof. The proof is based on observations similar to the ones in [16, Proof of Lemma 11]. Assume that $Y$ intersects both $X$ and $\mathcal{R}$. Since $|Y| \geq 3$, we can take distinct elements $x, y, z \in Y$ with $x \in X$, $z \in \mathcal{R}$ and $y$ belonging to either $X$ or $\mathcal{R}$. Suppose first that $y \not\in N[z]$. Let $v \in V(H_G)$ be such that $N[v] \cap \{x, y, z\} = \{y, z\}$. Since $y \not\in N[z]$ and $\mathcal{R}$ is a stable set, necessarily $v \in X$. But then, $x \in N[v]$ because $X$ is a clique, thus contradicting that $N[v] \cap \{x, y, z\} = \{y, z\}$. As a result, we must have $y \in N[z]$. Let $u \in V(H_G)$ be such that $N[u] \cap \{x, y, z\} = \{x, z\}$. Since $u, y \in N[z]$ and $N[z]$ is a clique, necessarily $y \in N[u]$. Again, the latter contradicts our assumption that $N[u] \cap \{x, y, z\} = \{x, z\}$.

First we suppose that $Y \subseteq X$. Since we assumed that the VC-dimension of $H_G$ is strictly less than $k$, $Y$ is not shattered in $H_G$. Thus

there exists $Z \subseteq Y$ that is not a trace of any range in $\mathcal{R}$.  

(1)

Since $Y$ is shattered by $H_G$, there is a vertex $v_Z \in V(H_G)$ such that $N[v_Z] \cap Y = Z$. The definition of $H_G$ and (1) imply that there is no $v \in \mathcal{R}$ such that its neighborhood in $H_G$ satisfies $N[v] \cap Y = Z$ and so we get that $v_Z \in X$. Since $H_G[X]$ is a clique, we conclude that $Z = Y$. However, there exists a range in $\mathcal{R}_3$ which contains the entire set $Y$ (namely, the one corresponding to the full set $S = \{1, 2, \ldots, k\}$), a contradiction with (1).

Finally, we consider the case of $Y \subseteq \mathcal{R}$. Let $y \in Y$ be any vertex. Since $Y$ is shattered, each of the $2^k - 1$ subsets of $Y$ that contain $y$ is either the trace of $N[y]$ or the trace of $N[x]$ for some neighbor $x$ of $y$. Therefore, $|N[y]| \geq 2^{k-1} \geq 4$, and so the range in $H_G$ corresponding to $y$ has size at least 3, in particular, $y \in \mathcal{R}_3$. For any $Y \subseteq \mathcal{R}_3$, the closed neighborhoods of the vertices of $H_G$ can have the following traces on $Y$:
if \( v \in \mathcal{R} \), then \( N[v] \cap Y \) is equal to \( \{ v \} \) if \( v \in Y \) and \( \emptyset \) otherwise;

if \( v = (x, i) \in X \), then \( N[v] \cap Y \) contains those vertices of \( Y \) that correspond to index sets \( S \) with \( i \in S \) (see the definition of \( \mathcal{R}_3 \)).

That is, the neighborhoods of vertices in \( \mathcal{R} \) can only induce the empty set and the \( k \) singleton traces on \( Y \), and for any \( x, y \in V(G) \) and \( i \in \{ 1, 2, \ldots, k \} \), we have \( N[(x, i)] \cap Y = N[(y, i)] \cap Y \). This implies that the number of vertices in \( Y \) that have pairwise different neighborhoods in \( Y \) is at most \( k \). On the other hand, since \( Y \) is shattered, we need to obtain each of the \( 2^k \) subsets of \( Y \) as a trace, which implies that \( 2^k \leq k + k + 1 \), and thus \( k \leq 2 \), a contradiction. ▶

### B Simple bounds

**Lemma 7.** A \( k \)-degenerate graph has VC-dimension at most \( k + 1 \), and this bound is sharp.

**Proof.** Let \( G = (V, E) \) be a \( k \)-degenerate graph and consider a shattered set \( X \subseteq V \). Let \( Z = X \cup \{ v_Y \mid Y \subseteq X \} \), where \( v_Y \) denotes an arbitrary vertex such that \( N[v_Y] \cap X = Y \). We consider the induced subgraph \( G[Z] \) and we iteratively remove all vertices of \( Z \setminus X \) with at most \( k \) neighbours. Let \( Z' \) be the set of remaining vertices. Note that \( X \subseteq Z' \). Since \( G[Z'] \) is also \( k \)-degenerate, it has some vertex \( x \) with at most \( k \) neighbours. Since we iteratively removed all vertices with at most \( k \) neighbours in \( Z \setminus X \), then necessarily \( x \in X \). Furthermore, all vertices of \( Z \setminus Z' \) must be of the form \( v_Y \) for some \( |Y| \leq k \). As a result, the number of neighbours of \( x \) in \( G[Z] \) is no more than \( k + \sum_{i=0}^{k-1} \binom{|X| - 1}{i} \). However, since \( X \) is shattered, and there are \( 2^{|X| - 1} \) subsets of \( X \) containing vertex \( x \), we must have \( |N[x] \cap Z| \geq 2^{|X| - 1} \). In particular, the number of neighbours of \( x \) in \( G[Z] \) must be at least \( 2^{|X| - 1} - 1 \). Suppose by contradiction that \( k < |X| - 1 \). Then,

\[
k + \sum_{i=0}^{k-1} \binom{|X| - 1}{i} = k + 2^{|X| - 1} - \sum_{i=k}^{|X| - 1} \binom{|X| - 1}{i}
\leq k + 2^{|X| - 1} - \sum_{i=|X| - 2}^{|X| - 1} \binom{|X| - 1}{i}
= k + 2^{|X| - 1} - \binom{|X| - 1}{|X| - 2} - \binom{|X| - 1}{|X| - 1}
= k + 2^{|X| - 1} - |X|
< 2^{|X| - 1} - 1
\]

A contradiction. Hence, \( |X| \leq k + 1 \). This is sharp for \( k = 1 \) because trees are 1-degenerate, and there exist 2-shattered subsets in trees (e.g., any two leaves in a star with at least three leaves). ▶

Finally, we show upper bounds on the VC-dimension in terms of sizes of maximum and maximal matchings.

**Lemma 8.** Let \( G = (V, E) \) be a non-empty graph and \( M \) be a maximal matching of \( G \). Then the VC-dimension of \( G \) is at most \( 2|M| \). Moreover, if \( \nu(G) \) is the size of a maximum matching in \( G \), then we have \( \text{VCDim}(G) \leq \nu(G) + 1 \).

**Proof.** Since \( G \) is non-empty, any maximal matching has at least one edge and thus the statements trivially hold if \( \text{VCDim}(G) \leq 2 \). Let \( X \subseteq V \) be a shattered set of size \( \text{VCDim}(G) \). If \( M \) covers every vertex of \( X \), then we have \( |M| \geq \frac{1}{2} \cdot \text{VCDim}(G) \). Assume that there exists \( x \in X \) which is not covered by \( M \). Since \( M \) is maximal, each of the at least \( 2^{\text{VCDim}(G) - 1} - 1 \) neighbors of \( x \) need to be covered by \( M \), which implies \( |M| \geq \frac{1}{2} \cdot (2^{\text{VCDim}(G) - 1} - 1) \geq \frac{1}{2} \cdot \text{VCDim}(G) \) for any graph with \( \text{VCDim}(G) \geq 3 \).
Table 4 VC-dimension lower bounds computed with \( \text{maxvisits} = 16, 32, 64, 128, 256 \) (bold values indicate that the bound matches the exact value), and the corresponding execution time, the “read” column corresponds to the time for reading the graph.

<table>
<thead>
<tr>
<th>Graph</th>
<th>VC-dim</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>read</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-MV-Physical-3.5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0.00</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.10</td>
</tr>
<tr>
<td>BIO-SYS-Aff-Cap-MS-3.5</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>0.15</td>
<td>0.18</td>
<td>0.18</td>
<td>0.20</td>
<td>0.20</td>
<td>0.23</td>
</tr>
<tr>
<td>BIO-SYS-Aff-Cap-RNA-3.5</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>0.02</td>
<td>0.03</td>
<td>0.02</td>
<td>0.03</td>
<td>0.03</td>
<td>0.04</td>
</tr>
<tr>
<td>dp20170205</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0.04</td>
<td>0.05</td>
<td>0.05</td>
<td>0.06</td>
<td>0.07</td>
<td>0.07</td>
</tr>
<tr>
<td>oregon2-010331</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.04</td>
</tr>
<tr>
<td>CAIDA-as-20130601</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>0.07</td>
<td>0.09</td>
<td>0.09</td>
<td>0.11</td>
<td>0.13</td>
<td>0.13</td>
</tr>
<tr>
<td>DIMES-201204</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>0.04</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.06</td>
<td>0.06</td>
</tr>
<tr>
<td>as-skitter</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>6.11</td>
<td>7.04</td>
<td>7.42</td>
<td>7.98</td>
<td>8.24</td>
<td>9.92</td>
</tr>
<tr>
<td>p2p-Gnutella09</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.03</td>
</tr>
<tr>
<td>gnutella31</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0.12</td>
<td>0.11</td>
<td>0.11</td>
<td>0.13</td>
<td>0.18</td>
<td>0.26</td>
</tr>
<tr>
<td>notreDame</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>0.74</td>
<td>1.05</td>
<td>1.16</td>
<td>1.12</td>
<td>1.19</td>
<td>1.20</td>
</tr>
<tr>
<td>y-BerkStan</td>
<td>7</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>3.49</td>
<td>4.87</td>
<td>4.88</td>
<td>5.08</td>
<td>5.39</td>
<td>5.88</td>
</tr>
<tr>
<td>ca-HepPh</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0.06</td>
<td>0.07</td>
<td>0.10</td>
<td>0.08</td>
<td>0.10</td>
<td>0.14</td>
</tr>
<tr>
<td>com-dblp</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0.58</td>
<td>0.71</td>
<td>0.78</td>
<td>0.81</td>
<td>0.95</td>
<td>0.99</td>
</tr>
<tr>
<td>epinions1</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>0.25</td>
<td>0.33</td>
<td>0.35</td>
<td>0.36</td>
<td>0.40</td>
<td>0.46</td>
</tr>
<tr>
<td>facebook-combined</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>0.04</td>
<td>0.04</td>
<td>0.05</td>
<td>0.06</td>
<td>0.07</td>
<td>0.10</td>
</tr>
<tr>
<td>twitter-combined</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>0.64</td>
<td>0.70</td>
<td>0.70</td>
<td>0.82</td>
<td>1.03</td>
<td>1.45</td>
</tr>
</tbody>
</table>

To show that \( \text{VCdim}(G) \leq \nu(G) + 1 \), it is sufficient to construct a matching where \(|X| - 1\) vertices of \(X\) are matched to a vertex outside of \(X\). Since \(X\) is shattered, for any \(x \in X\), there exists a vertex \(v_x \in V(G)\) such that \(N[v_x] \cap X = \{x\}\). Observe that we have either \(v_x \not\in X\) or \(v_x = x\) and the second option is only possible if \(x\) has no neighbors in \(X\). We build a matching \(M\) of size \(|X| - 1\) as follows. First we add all edges \(\{x, v_x\}\) to \(M\) where \(x \in X\) is such that it has at least one neighbor in \(X\). After this, if there is a remaining set \(Y \subseteq X\) which is not yet covered by \(M\), then \(Y\) has to be a stable set. Since \(X\) is shattered, there is a vertex \(v_Y\) such that \(v_Y \cap X = Y\). As \(Y\) is a stable set, \(v_Y \not\in Y\) if \(|Y| \geq 2\). Thus, we can cover each element of \(Y\) (except maybe one) by taking any \(y \in Y\), adding \(\{v_Y, y\}\) to \(M\), and recursing on \(Y' = Y \setminus \{y\}\). In the end, we get a matching of size \(|M| \geq |X| + |Y| - 1 = \text{VCdim}(G) - 1\). \(\blacktriangle\)

Note that stars with at least three leaves have matching number one and VC-dimension two, thus the bound of Lemma 8 is sharp.

C Experiments

C.1 Lower-bound computation

Table 4 gives the lower-bounds obtained by our lower-bound heuristic for various values of \(\text{maxvisits}\) while \(KBG\) uses \(\text{maxvisits} = 64\). It also provides the corresponding running times for reading the graph and computing the lower-bound. As a reference, column “read” indicates the time spent for just reading the graph.
We observe a tradeoff where increasing \textit{maxvisits} provides generally a better lower-bound at the cost of a longer running time. Note the exception of twitter-combined for which the best bound is obtained only for \textit{maxvisits} = 32. We also note that the running time stays within a factor 5 of the time taken for reading the graph, even for \textit{maxvisits} = 256. The exception of twitter-combined let us think that there is room for improvement of the tuning of our heuristic. For example, the choice of \textit{maxvisits}/2 for limiting the for loop of \textsc{ExploreShattered} was not intensively explored. However, it already provides lower-bounds which are often exact or one less than the true VC-dimension. This is indeed the case for \textit{maxvisits} = 128, and almost the case for \textit{maxvisits} = 64 where notreDame is the only exception with a lower-bound which is two less than the VC-dimension.

### C.2 Upper bounds

Table 5 lists the upper-bounds we can quickly compute on our dataset. The degree upper-bound \(\lfloor \log \Delta \rfloor + 1\) where \(\Delta\) is the maximum degree appears to always be the best one. However, it can be as large as twice the true value and gives a poor confidence bound compared to what we obtained for lower-bounds. The node upper-bound \(\lfloor \log n \rfloor\) where \(n\) is the number of nodes is almost always greater. The only graph where it matches the degree upper-bound is facebook-combined which has high maximum degree \(\Delta\) compared to its number \(n\) of nodes as it satisfies \(\Delta > n/4\) (see Table 1). Finally, the degeneracy upper-bound appears to be good on graphs with low degree, that is road networks and grid like graphs, while it can be very high for graphs with many high degree nodes. Note that it can be quite high even for graphs with relatively low VC-dimension such as ca-HepPh.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
graph & lb64 & VDim & \(\lfloor \log \Delta \rfloor + 1\) & \(\lfloor \log n \rfloor\) & degen+1 \\
\hline
BIO-MV-Physical-3.5 & 5 & 5 & 11 & 14 & 37 \\
BIO-SYS-Aff-Cap-MS-3.5 & 7 & 7 & 12 & 15 & 52 \\
BIO-SYS-Aff-Cap-RNA-3.5 & 6 & 7 & 12 & 13 & 55 \\
dip20170205 & 5 & 5 & 9 & 14 & 22 \\
\hline
oregon2-010331 & 5 & 6 & 12 & 13 & 32 \\
CAIDA-as-20130601 & 6 & 7 & 12 & 15 & 69 \\
DIMES-201204 & 7 & 7 & 12 & 14 & 35 \\
ass-skitter & 8 & 8 & 16 & 20 & 112 \\
\hline
p2p-Gnutella09 & 5 & 5 & 7 & 12 & 11 \\
gnutella31 & 3 & 4 & 7 & 15 & 7 \\
notreDame & 4 & 6 & 14 & 18 & 156 \\
y-BerkStan & 7 & 7 & 17 & 19 & 202 \\
\hline
cia-HepPh & 5 & 5 & 9 & 13 & 239 \\
com-dblp & 4 & 5 & 9 & 18 & 114 \\
opinionsnlp & 7 & 7 & 12 & 16 & 68 \\
facebook-combined & 6 & 6 & 11 & 11 & 116 \\
twitter-combined & 6 & 7 & 12 & 16 & 97 \\
\hline
\end{tabular}
\caption{Some upper-bounds of Lemma 4 compared to lower-bounds and true VC-dimension.}
\end{table}
Top-k Frequent Patterns in Streams and Parameterized-Space LZ Compression

Patrick Dinklage
TU Dortmund University, Germany

Johannes Fischer
TU Dortmund University, Germany

Nicola Prezza
Ca’ Foscari University of Venice, Italy

Abstract
We present novel online approximations of the Lempel-Ziv 77 (LZ77) and Lempel-Ziv 78 (LZ78) compression schemes [Lempel & Ziv, 1977/1978] with parameterizable space usage based on estimating which k patterns occur the most frequently in the streamed input for parameter k. This new approach overcomes the issue of finding only local repetitions, which is a natural limitation of algorithms that compress using a sliding window or by partitioning the input into blocks. For this, we introduce the top-k trie, a summary for maintaining online the top-k frequent consecutive patterns in a stream of characters based on a combination of the Lempel-Ziv 78 compression scheme and the Misra-Gries algorithm for frequent item estimation in streams. Using straightforward encoding, our implementations yield compression ratios (output over input size) competitive with established general-purpose LZ-based compression utilities such as gzip or xz.

2012 ACM Subject Classification Theory of computation → Data compression; Theory of computation → Pattern matching; Theory of computation → Sketching and sampling

Keywords and phrases compression, streaming, heavy hitters, algorithm engineering

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.9

Supplementary Material Software (Source Code): https://github.com/pdinklag/top-k-compress
archived at swh:1:dir:c586220ee1ae265be3a77be75b09f2fdcb55843c

Funding Patrick Dinklage: Funded by the Deutsche Forschungsgemeinschaft (DFG) under the Research Grants programme (project No. 501086801). Johannes Fischer: Funded by the Deutsche Forschungsgemeinschaft (DFG) under the Research Grants programme (project No. 501086801). Nicola Prezza: Funded by the European Union (ERC, REGINDEX, 101039208). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

Acknowledgements The authors gratefully acknowledge the computing time provided on the Linux HPC cluster at Technical University Dortmund (LiDO3), partially funded in the course of the Large-Scale Equipment Initiative by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as project 271512359.

1 Introduction

Lempel-Ziv schemes are arguably among the most popular both in theory and practice of compressing data. The main idea is to identify repeating patterns and replace them by references to other occurrences. Compression is achieved if these references can be encoded with less bits than the sequence they replace.
Top-k Frequent Patterns in Streams and Parameterized-Space LZ Compression

The memory requirement for computing Lempel-Ziv parsings, however, is a major practical issue. If inputs become large, linear dependency on the input or output size becomes prohibitive. One of the most common approaches to tackle this issue is by processing the input using a sliding window: the memory requirement then depends only on the (parameterizable) window size. As a trade-off, the detection of repetitions is limited to within the window. Despite this, sliding windows back popular everyday compression utilities such as gzip (very small window) or xz/7-zip (arbitrarily large window).

In this work, we attempt to alleviate the issue of being able to find only local repetitions and instead get a global sense of repeating patterns, all while keeping the space requirement parameterizable. The key to our approach is the notion of heavy hitters: if we can estimate which k patterns are the most frequent at any time while streaming the input, that helps us find repetitions globally. Because the patterns are frequent, the hope is that they also contribute the most to compression. We can encode them as references into a universe of size k, i.e., an index into the k currently frequent patterns. The produced output can then be decoded by following the same protocol. Because we never account for more than k patterns, the parameter k directly controls our space consumption.

Related Work

Mining frequent items in a stream has a long history of research. Aggarwal and Yu [2] give a survey on established techniques to estimate the frequencies of items. In this work, we use and adapt the algorithm due to Misra and Gries [28] and the Space-Saving data structure due to Metwally et al. [27] to estimate online the k most frequent items (outlined in section 2.1).

Maruyama and Tabei [25] use an idea similar to our work for grammar compression. Their algorithm computes online and in \(O(k)\) space a straight-line program that produces the input. They explore methods for estimating online the k most frequent production rules, in the hope that these contribute the most for compression, using techniques similar to what we describe in section 3. On large genomic sequences (hundreds of gigabytes), their approach achieves good compression rates with impressively short running times. However, they make heavy use of the fact that genomic sequences have a very small alphabet of nucleotide bases, and it is not clear how their algorithm performs for larger alphabets. Another related result is that of De Agostino [3], who considered a constrained-dictionary version of the Lempel-Ziv 78 scheme (LZ78). The dictionary is limited to size k and if it is full, a heuristic is used to determine which entry to replace next, with the focus being on the last recently used (LRU) heuristic and variants. Their experiments targeted inputs of few megabytes and very small dictionaries, and they did not focus on performance other than the constrained memory consumption. To that end, their results are difficult to project to larger inputs. The general idea of constraining the size of the LZ78 dictionary, however, matches our work. Apart from this, there has been work on finding trade-offs between the running time and space consumption of LZ78 parsers, Arroyuelo et al. [6] give a recent overview.

Apart from these, there has been work on identifying frequent itemset subsequences, Gan et al. give a survey in [17]. The problem is much more general than what we consider, as (1) the subsequences need not be consecutive in the stream and (2) each item in a subsequence may be any one from a set. Apostolico et al. [5] use mining techniques to find so-called motifs later used for Lempel-Ziv-Welch compression. Motifs can roughly be considered strings with wildcards, and thus their scenario is more related to mining frequent subsequences than it is to mining frequent consecutive patterns. The bioinformatics community has done work on estimating the number of distinct k-mers, or a histogram thereof, in streamed genomic sequences [26, 29, 31, 8] (here, k refers to the fixed length of relevant substrings and is not
to be confused with top-k). However, their algorithms are not concerned about what are the most frequent k-mers. Furthermore, they aim at a fixed pattern length k and on a known constant-size alphabet, properties that cannot be exploited without loss of generality. Fischer et al. [14] consider the problem of string mining under frequency constraints, which is more similar to our scenario. Their solution is optimal in terms of accuracy, but it requires a full-text index and therefore disqualifies in an online scenario. It furthermore finds patterns that have a given minimal support in the input, for which we know no direct translation to maintaining the k most frequent patterns.

Our Contributions

We introduce the top-k trie (section 3), a framework to estimate and maintain online the k most frequent patterns in a stream. It uses space at most $O(k)$ and requires constant amortized time per input character. We apply this to online approximations of the LZ78 (section 4) and LZ77 (section 5) parsings, which we implement and compare against established general-purpose compressors (section 6). Our algorithms are competitive in terms of speed and compression ratio albeit using a straightforward encoding. We finally analyze the impact of the parameter k on resource usage and the number of emitted phrases.

2 Preliminaries

Let $s \in \Sigma^*$ be a string over an alphabet $\Sigma$. For $i \in [1,|s|]$, we denote by $s[i]$ the i-th character of s. Let $j \in [1,|s|]$ and $j \geq i$, then $s[i..j]$ denotes the (consecutive) substring of characters from position i and j, both included, of s. For analysis, we use the word RAM model [18], where the memory consists of contiguous words of length $\Theta(n)$ bits each (by default, we state logarithms as base-2). We can access and perform arithmetic operations on a constant number of words in constant time. We further consider a streaming model, particularly the cash register model [30], where the input is a stream $S = x_1, x_2, \ldots, x_n$ of n items drawn from some domain $\Gamma$. The items are received one by one and in non-rewindable fashion. There is no previous knowledge about the length $n \to \infty$ of the stream, and it may far exceed the size $M$ of available memory. To that end, it is not possible to store $S$ in memory for random access. Furthermore, nothing is known about $\Gamma$, so even storing information about all $\sigma \leq n$ distinct items that occur in $S$ is infeasible (e.g., a histogram – see also [4]). However, we are allowed to store pointers into the stream or count the number of occurrences of a selected set of items. In that regard, we assume $M = \Omega(\text{polylog } n)$.

2.1 The Misra-Gries algorithm for top-k frequent item estimation

For an item $x \in \Gamma$, let $f_x \in [0,n]$ be its frequency, i.e., the number of occurrences of $x$ in $S$. We consider the problem of finding the top-k most frequent items from $\Gamma$ that occur in $S$. Formally, we want to find a set $F$ with $|F| = k$ and $\forall x \in F : f_x \geq \max\{f_y \mid y \in \Gamma \setminus F\}$. As noted earlier, we cannot hope to maintain a histogram of frequencies for all the items of $\Gamma$.

The Misra-Gries algorithm [28] computes a summary of $F$ in space $O(k)$. For some $x \in \Gamma$, we call $f'_x$ the estimated frequency of $x$ in $S$. Before reading from $S$, we initialize $F := \emptyset$. After reading the next item $x$ from $S$, we act as follows: if $x \in F$, we increment the estimated frequency $f'_x := f'_x + 1$. If $x \notin F$ and $|F| < k$, we insert $x$ into $F$ with $f'_x := 1$. Otherwise ($x \notin F \land |F| = k$), we decrement the estimated frequencies $f'_y := f'_y - 1$ for all $y \in F$ and then delete any $y$ from $F$ for which $f'_y = 0$. 


It is easy to see that $f'_s \leq f_s$ holds for any $x \in \Gamma$, that is, the Misra-Gries algorithm underestimates frequencies. As shown in [9], it holds that $f'_z \geq f_x - n/k$. The algorithm is fast in practice, even though it has a relatively high false positive rate [10, referred to as \textsc{Frequent}]. For this work, however, false positives are not a serious disadvantage.

### 2.2 The Space-Saving Data Structure

The Space-Saving data structure of [27], even though proposed independently, can be used to implement the Misra-Gries algorithm. For each distinct estimated frequency $f'$ of the items in $F$, it maintains a bucket. Each bucket holds a doubly-linked list of the items $x \in F$ such that $f'_x = f'$ in arbitrary order. The buckets are also maintained in a doubly-linked list ordered by their represented frequencies $f'$. Figure 1 shows an example.

The data structure requires space at most $O(k)$ as $|F| = k$. Creating, deleting, inserting items into or deleting items from a bucket each takes constant time thanks to the use of doubly-linked lists. Thus, a new item $x$ with $f'_x = 1$ can be inserted in constant time by inserting it into the first bucket, and the frequency of an item $y$ can be incremented in constant time by removing it from bucket $f'_x$ and inserting it into bucket $f'_y + 1$.

For the Misra-Gries algorithm, we require an operation to decrement all frequencies. This can be simulated by maintaining a threshold $\theta$ that is initially zero [1]. Let $x \notin F$ and $|F| = k$. Instead of decrementing all frequencies, we first test whether there is any item $y$ with $f'_y \leq \theta$. This can be done in constant time by testing whether the minimum bucket represents a frequency $f' \leq \theta$. If that is the case, we delete an arbitrary item $y$ contained in the bucket and recycle its entry for $x$, inserting it with $f'_y := \theta$. Otherwise, we increment $\theta := \theta + 1$. This lazy approach simulates the decrement-all operation in constant time.

### 2.3 Tries

A trie [16] represents a set $\mathcal{S}$ of $m$ distinct strings in a tree as follows. The root represents the empty string $\epsilon$. For every string $s \in \mathcal{S}$, there is a path $\pi_s = v_0, v_1, \ldots, v_{|s|}$ of nodes such that $v_0$ is the root and each node $v_i$ represents the prefix $s[1..i]$. To that end, the edge from node $v_{i-1}$ to $v_i$ is labeled by $s[i]$, and thus $s$ is the concatenation of the edge labels along $\pi_s$. The total number of nodes in the trie is at most $1 + \sum_{s \in \mathcal{S}} |s|$.

### 2.4 Lempel-Ziv 78

The Lempel-Ziv 78 (LZ78) compression scheme [35] factorizes the input $S \in \Sigma^*$ into $z$ phrases $f_1, \ldots, f_z$ such that $f_1 \cdots f_z = S$. The $i$-th phrase ($1 \leq i \leq z$) is $f_i = f_j \alpha$ for $j < i$ and $\alpha \in \Sigma$ such that $f_j$ is the longest possible previous phrase such that $f_j \alpha$ does not occur in $f_1 \cdots f_{i-1}$. In case no such $j$ exists, we say $f_j = f_0 := \epsilon$ and $f_i = \alpha$ denotes a new character.

Standard algorithms to compute the LZ78 parsing use a trie $T$ where each node represents a phrase and is labeled by the phrase number. Figure 2 shows an example. Suppose that we have already computed the first $i-1$ phrases. We memorize the current node number $j$, which is initially the root (representing $j = 0$ and the empty phrase $\epsilon$). When we read the next character $\alpha$ from $S$, we attempt to follow the edge at node $j$ labeled $\alpha$. If that edge
exists, we set $j$ to the number of the connected node. Otherwise, node $j$ spells out $f_j$, the longest phrase such that $f_j\alpha$ has no previous occurrence in $S$, and thus the next phrase is $f_j\alpha$, which we encode as the tuple $(j, \alpha)$. We create a new child node $i$ and connect it to $j$ with an edge labeled $\alpha$. Then, we reset $j := 0$ back to the root to begin with the next phrase.

This process is repeated until $S$ has been fully parsed.

### 2.5 Lempel-Ziv 77

The Lempel-Ziv 77 (LZ77) compression scheme \cite{LZ77} factorizes the input $S \in \Sigma^*$ into $z$ phrases $f_1, \ldots, f_z$ such that $f_1 \cdots f_z = S$. The $i$-th phrase $f_i$ is either the first occurrence of some character $\alpha = f_i$ in $S$, or it is the longest possible prefix of $S[[f_1 \cdots f_{i-1}] + 1 .. n]$ that already occurs previously in $S[1 .. [f_1 \cdots f_i]]$.

One key difference to LZ78 is that LZ77 phrases may overlap. Consider, as an example, the input $S = \alpha^6$ for some character $\alpha$: while the LZ78 parsing of $S$ consists of the three phrases $\alpha$, $\alpha^2$ and $\alpha^3$, the LZ77 parsing consists only of the two phrases $f_1 = \alpha$ and $f_2 = \alpha^5$. As a result of overlaps, LZ77 parsings typically consist of fewer phrases than LZ78 parsings for the same input, which in turn allows for smaller encodings of the compressed input.

However, compared to LZ78, computing the LZ77 parsing is more complicated. Finding the next LZ77 phrase means finding the longest previous occurrence of a prefix of the remaining input. Algorithms to compute LZ77 therefore typically make use of some kind of (possibly compressed) full-text index. In appendix C, we describe the well-known algorithm that computes the LZ77 parsing in time and space $O(n)$ via the suffix array of the input. More recent examples that improve on the time and/or space requirement include \cite{LZ77-RecentExamples}.

### Approximations

The requirement of space linear in the input or output size becomes prohibitive in practice when the size of the input or the index data structure exceeds the available memory. It is therefore straightforward to think about a trade-off where only an approximation of LZ77 is computed in smaller – preferably parameterized – space at the cost of worse compression. Myriad approximation algorithms have been proposed that compute so-called LZ-like parsings via hashing \cite{LZ-LikeHashing}, references \cite{LZ-LikeReferences} or samplings with synchronizing properties \cite{LZ-LikeSamplings}, to name but a few examples. Considering everyday compression utilities such as gzip or xz, however, the arguably most popular strategy involves using a sliding window of fixed size $w$. Here, the requirement of finding a longest previous occurrence is relaxed to a search only within the window. While the space is then governed only by parameter $w$, the number of produced phrases may increase dramatically. A common way to counter this is to encode phrases using sophisticated encoding schemes (e.g., DEFLATE \cite{DEFLATE} or LZMA\footnote{To the best of our knowledge, there is no official specification of LZMA, but a de-facto specification of the XZ format is available at https://tukaani.org/xz/xz-file-format.txt}).
Algorithm 1 Framework for maintaining online the top-$k$ trie $T_F$ for an input stream.

In this listing, the trie is modeled as a set of node numbers where 0 denotes the root node.

Input: Stream $S \in \Sigma^*$, parameter $k \in \mathbb{N}$

Output: Top-$k$ trie $T_F$ (maintained online)

1. $T_F \leftarrow \{0\}$; $\theta \leftarrow 0$, $v \leftarrow 0$

2. while there is another character $\alpha$ on $S$ do

   3. if $v$ has a child $u$ with edge labeled $\alpha$ then

      4. $f'_u \leftarrow f'_u + 1$

      5. $v \leftarrow u$

   6. else if $|T_F| < k$ then

      7. $T_F \leftarrow T_F \cup \{\text{new node } u\}$  \hspace{1cm} // grow trie

      8. $f'_u \leftarrow \theta + 1$

      9. make $u$ a child of $v$ with edge labeled $\alpha$

   10. else if there is a leaf $u$ in $T_F$ with $f'_u \leq \theta$ then

       11. delete the edge to $u$ \hspace{1cm} // recycle non-frequent

       12. $f'_u \leftarrow \theta + 1$

       13. make $u$ a child of $v$ with edge labeled $\alpha$

   14. else

       15. $\theta \leftarrow \theta + 1$ \hspace{1cm} // decrement all

       16. $v \leftarrow 0$

3. The Top-$k$ Trie for Frequent Pattern Estimation

Consider a stream $S \in \Sigma^*$ of characters from the alphabet $\Sigma$ and the problem of maintaining online the $k$ most frequent substrings (consecutive patterns of characters) occurring in $S$. Even if the alphabet is small (e.g., a byte alphabet, an ASCII alphabet or even nucleotide bases from a DNA sequence), the number of substrings is quadratic in the number of characters read from $S$ and thus maintaining a histogram of frequencies for all patterns is impractical. Instead, this motivates the use of a summary.

We propose a variation of the lazy Misra-Gries algorithm that maintains the currently frequent patterns in a trie $T_F$ of size at most $k$, the Space-Saving data structure containing the nodes of $T_F$ and a threshold $\theta$. At each node $v \in [0,k]$ in $T_F$, we store the estimated frequency $f'_v$ of the pattern spelled out by the edge labels on the path from the root to $v$, as well as a back link into the Space-Saving data structure for constant-time access. In the remainder of this article, we refer to this data structure as the top-$k$ trie. Because $T_F$ has at most $k$ nodes and the Space-Saving data structure can only have $k$ distinct buckets holding a total of $k$ entries, the space of the top-$k$ trie is $O(k)$.

We process the input stream $S$ as shown in algorithm 1. Initially, $T_F$ consists of only the root numbered 0. We call $v$ the current node, initially the root. Upon reading the next character $\alpha$ from $S$, we find the edge labeled $\alpha$ from $v$ to a child $u$. If the edge exists, we increment $f'_u$ and continue with $v := u$. Otherwise, if $T_F$ is not yet full ($|T_F| < k$, line 7), we create a new leaf $u$ and make it a child of $v$ with edge labeled $\alpha$. If $T_F$ is full, let $u$ be a leaf (not an inner node, as we will discuss later) with $f'_u \leq \theta$ that we find using the Space-Saving data structure. If $u$ exists, we recycle it by severing its edges (from its parent and to its children) and making it a child of $v$ with edge label $\alpha$ (line 11). Otherwise, if there is no node to recycle, we increment the threshold $\theta$ to simulate the decrementation for all frequent patterns (line 15). Unless we can follow an existing edge, we reset the current node to the root (line 17). Whenever a frequency $f'_u$ is modified for some node $u$, we update the Space-Saving data structure accordingly.
There is a very apparent similarity between algorithm 1 and the LZ78 compression scheme (section 2.4). Our algorithm makes implicit use of the intuitive correlation between frequent patterns and Lempel-Ziv phrases, which capture repeating patterns. In section 4, we show how the top-k trie can be used to approximate LZ78.

**Preventing Orphans by Recycling Leaves**

The recycling of a node in algorithm 1 comes with a complication: u may be an inner node with up to $O(k)$ children that become orphans as a result of the operation. If the represented pattern was to be inserted back due to becoming frequent again, to ensure integrity, the orphaned former children that still remain would have to be connected back. However, this is non-trivial, because the node number may now be other than u.

A pragmatic solution would be to delete the entire subtrie of u from $T_F$ if u is to be recycled. This would be safe in the sense that we cannot lose any frequent patterns, because the represented string is a prefix of all strings represented in the subtrie, and thus their frequencies are at most $f_u' \leq \theta$. However, this would take time $O(k)$.

Instead, we ensure that only the leaves of $T_F$ can ever be recycled. To achieve this, only the leaves are maintained in the Space-Saving data structure to begin with. (Besides preventing orphans, this yields practical speedups because for realistic inputs, a vast majority of nodes in $T_F$ are inner nodes. Avoiding the operations for incrementing the frequencies of inner nodes, albeit constant time, saves a considerable amount of work.) This requires us to consider two new operations in the Space-Saving data structure. First, whenever a leaf $v$ becomes a leaf because its last child has been recycled, we need to insert it into the Space-Saving data structure. The insert frequency $f'_v$ can be arbitrarily large, and finding the bucket into which to insert v would require up to $O(k)$ steps in a doubly-linked list of buckets. We address this in the following.

**Frequency Limitation and Renormalization for Fast Bucket Access**

Instead of maintaining the buckets of the Space-Saving data structure in a doubly-linked list, we propose to maintain them in an array of fixed size $f_{\text{max}} = \Theta(k)$ where the head of the bucket for frequency $f'$ is stored in the $f'$-th entry. Trivially, this requires space $O(f_{\text{max}}) = O(k)$. Along with this change, we introduce two constraints that ensure that estimated frequencies always fall in the range $[0, f'_{\text{max}}]$ independent of the input stream:

1. We do not any increment a frequency $f'_v = f'_{\text{max}}$ (for some node v) any further.
2. Whenever $\theta$ is incremented to $f'_{\text{max}} / 2$, we renormalize the data structure by setting $f'_v := f'_v - f'_{\text{max}} / 2$ for all nodes $v$ of $T_F$ and reset $\theta := 0$. (Note that $\theta$ is only incremented if there is no node $u$ with $f'_u \leq \theta$. Thus, $f'_v > \theta$ holds before and after renormalization.)

The renormalization introduced by constraint 2 takes time $O(k)$, but amortizes to constant time since at least $\Theta(f'_{\text{max}}) = \Theta(k)$ incrementations are needed for another renormalization to be required. In actuality, let us recall how $T_F$ behaves a lot like the LZ78 trie (see section 4). The only occasions at which $\theta$ may be incremented is when a leaf of $T_F$ has been reached and algorithm 1 attempts to follow an edge that does not exist. Let $z$ be the number of LZ78 phrases of $S$, then there are only ever $O(z)$ occasions at which $\theta$ may be incremented. Thus, renormalizations are expected to occur rarely if $S$ is repetitive and/or $f'_{\text{max}}$ is sufficiently large. In our experiments on inputs of size 100 GiB (section 6), we never observed even one renormalization. This practical observation also justifies neglecting the bias introduced.
by constraint 1, which causes very frequent patterns to be underestimated more than less frequent patterns. For this underestimation to cause some pattern \( u \) of frequency \( f_u > f'_{\text{max}} \) to fade out of \( T_F \), at least one renormalization would have to occur and then, additionally, a sufficiently long substring of \( S \) with no occurrence of \( u \) would have to follow.

**Error in Frequency Estimation**

We note\(^2\) that the estimated frequency of some frequent pattern \( u \) is actually the sum of the frequencies stored in the subtrie of the node corresponding to \( u \) in \( T_F \). For example, let \( S = \alpha^{10} \) for some \( \alpha \in \Sigma \) (and \( k \geq 4 \)): after processing \( S \) with algorithm 1, the frequency of stored at the node corresponding to \( \alpha \) will be only \( f'_{\alpha} = 4 \). The sum of the frequencies stored in its subtrie, however, is indeed 10.

It has been shown [9] that the underestimation of item frequencies by the Misra-Gries algorithm is bounded by \( n/k \). However, it is crucial to see that this is based on the assumption that every occurrence of an item on the stream is treated equally. That is not the case in our scenario: we do not account for all substrings of \( S \), but only a subset that is the result of a parsing similar to LZ78. It seems straightforward that this contributes to additional error in the Misra-Gries algorithm if considering the frequency of every substring of \( S \) individually, but leave open a detailed analysis that we conjecture to be non-trivial. However, we can still look at our running time improvements and how they contribute to the error of the underlying Misra-Gries structure under the assumption that all substrings are counted.

First, we established that we only recycle leaves from \( T_F \) and never inner nodes. This does not introduce any additional error, because the Misra-Gries algorithm cares not what item is removed as long as is has zero frequency. (In fact, in Misra-Gries, all items of zero frequency would be removed immediately.)

The limitation of estimated frequencies to \( f'_{\text{max}} \), on the other hand, introduces arbitrary error. Consider \( S = \alpha^n \) for some \( \alpha \in \Sigma \): there are \( n \) occurrences of \( \alpha \), but the highest estimated frequency will be \( \mathcal{O}(f'_{\text{max}}) = \mathcal{O}(k) \), and clearly it is \( \lim_{n \to \infty} n/k = \infty \). This corresponds to the underestimation bias against very frequent patterns that we discussed earlier. Even though the error is unbounded in the mentioned case, we found it to be negligible in practice for sufficiently large \( f'_{\text{max}} \).

The renormalization of frequencies does not contribute to any error directly. Recall how the threshold \( \theta \) simulates a marker for items that have zero estimated frequency. If \( \theta > 0 \), it means that in the original Misra-Gries algorithm, the frequencies of all items would have been decremented \( \theta \) times. To this end, the described renormalization can be considered a lazy application of these decrementation.

**Summary**

The top-\( k \) trie requires space \( \mathcal{O}(k) \) and allows processing each character from \( S \) in constant amortized time with a very unlikely worst case (renormalization). Even though it is not of interest in the following where we compress \( S \), the \( k \) patterns maintained by the algorithm could be enumerated via a traversal of \( T_F \).

\(^2\) We thank the anonymous reviewer to point out this fact.
4 Online Approximation of LZ78 in Parameterized Space

The similarity between the LZ78 and our top-\(k\) trie of section 3 should be very apparent at this point. We make it more explicit in a parameterized-space algorithm to approximate the LZ78 parsing of a stream \(S\) using the top-\(k\) trie. The algorithm can be seen as constrained-dictionary LZ78 similar to that of De Agostino [3], with the heuristic that only the \(k\) most frequently used phrases are maintained in the trie.

It works largely similar to the original LZ78 algorithm: let \(v\) be the current node in \(T_F\) (initially the root), and consider the next character \(a\) from \(S\). If there is an outgoing edge labeled \(a\) from \(v\), we follow that edge and set \(v\) to the number of the connected node. Otherwise, we output \(f_v a\) as the next phrase (e.g., by encoding the tuple \((v, a)\)). Whenever we visit a node or create a new node, we update the Space-Saving data structure as described in section 3. This algorithm is an immediate application of the top-\(k\) trie and therefore, it is easy to see that it requires \(O(k)\) space and amortized constant time per input character.

To decode the produced parsing, the decoder can simulate \(T_F\) like the encoder. For this, the parameter \(k\) must be stored at the beginning of the compressed output.

Let \(z\) be the number of LZ78 phrases and \(z'\) be the number of phrases produced by our parsing. It holds that \(z' \geq z\) for \(k \geq z\), the trie \(T_F\) is able to hold a node for every LZ78 phrase and it is \(z' = z\). For \(k < z\), our approximation produces more phrases. Consider a unary string \(a^n\) for some \(a \in \Sigma\). In this case, \(z = \Theta(\sqrt{n})\). For our approximation, because the size of \(T_F\) is limited to \(k\), it is \(z' = \Theta(n/k)\), and so the approximation ratio is \(z'/z = \Theta(\sqrt{n}/k)\). We conjecture that this also holds for arbitrary strings.

5 Online Approximation of LZ77 in Parameterized Space

We propose a new approximation of LZ77 that combines the top-\(k\) trie of section 3 with a blockwise computation of LZ77. We partition \(S\) into blocks of size \(B\) such that the \(i\)-th block is \(S_i := S[iB .. iB + B - 1]\). Let the top-\(k\) trie \(T_F\) be initially empty. After reading block \(S_i\) from \(S\), we first compute its LZ77 parsing (using the algorithm of appendix C) consisting of the \(z'\) phrases \(f_1, \ldots, f_{z'}\). Then, we compute a parsing of \(S_i\) consisting of \(\leq z'\) phrases as follows. Let \(m\) be the current position within \(S_i\) (initially \(m := 1\)). Let \(c := \min\left\{ \lfloor j | \lfloor f_1 \cdots f_j \rfloor \geq m \rfloor \right\}\) be the index of the LZ77 phrase that contains position \(m\). We call \(\ell := \lfloor f_1 \cdots f_c \rfloor - m\) the length of the remainder of \(f_c\). Furthermore, let \(s_m\) be the longest prefix of \(S_i[m .. B - 1]\) that can be spelled using the top-\(k\) trie \(T_F\) starting at the root. Now, if \(|s_m| > m - |f_1 \cdots f_c|\), then the next phrase in our parsing is \(g := s_m\) and we advance to position \(m := m + |s_m|\). Otherwise, we choose the phrase \(g := S_i[m .. m + \ell - 1]\) (the remainder of \(f_c\)) and advance to position \(m := m + \ell\). We then enter \(g\) into \(T_F\) in similar fashion as we parse the input in section 4 (LZ78): starting from the root, we navigate (and possibly insert) the edges of \(T_F\) using the characters from \(g\) as deep as possible and increment the frequencies of the corresponding prefixes.

To create an intuition, this algorithm attempts to get the best out of both worlds: it looks at the (remainder of the) next LZ77 phrase and the longest prefix that can be spelled out by \(T_F\), and then greedily picks whatever allows us to advance further in \(S_i\). This is visualized in figure 3. It is easy to see that the number of phrases produced this way is at most \(z'\), and may be less than \(z'\) if \(T_F\) contains useful strings. Doing this for all \([n/B]\) blocks results in an approximation of LZ77 with the total number of phrases being lower bounded by \(z\) (for \(B \geq n\) or suitable \(S\), we produce exactly \(z\) phrases).
We implemented the online approximations of LZ78 (topk-lz78, section 4) and LZ77 (topk-lz77, section 5) in C++20. The source code is publicly available as referred to on the front matter. In the experiment, we also consider blockwise-lz77, which computes the LZ77 parsing for every input block of size $B$ and then encodes it, i.e., without using the top-k trie. The variant blockwise-lz77+ simply uses a larger block size $B' > B$.

To be able to compare compression ratios, we implemented a blockwise encoding of the tuples representing LZ78 or LZ77 phrases, respectively, emitted by the compressors. For a parameter $b$, we buffer $b$ tuples in memory and write them to file once the buffer overflows. In a preliminary experiment, we found that $b := 32 \cdot 2^{10}$ gives the best overall results and thus use this value throughout our experiments. We encode tuples as follows.

- In topk-lz78, tuples are of the form $(j, \alpha)$ where $j < k$ is the number of a node in the top-$k$ trie and $\alpha \in \Sigma$ is the appended character. We encode $j$ in binary using $\lceil \log(k - 1) \rceil$ bits and $\alpha$ using Huffman codes according to the distribution among the $b$ buffered tuples.

- In topk-lz77, tuples are of the form $(\ell, \delta)$ as described in section 5. We encode $\ell$ using Huffman codes according to the distribution of all values $\ell$ among the $b$ buffered tuples. We exploit the practical observation that most referencing phrases are short: if $\ell \geq 255$,
Table 1 Listing of compressors that we compare against. For each compressor, we list the URL of its main website as well as the command line flags we use to execute it for best compression.

<table>
<thead>
<tr>
<th>Compressor</th>
<th>URL</th>
<th>Execution Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td><a href="http://gzip.org/">http://gzip.org/</a></td>
<td>-9</td>
</tr>
<tr>
<td>xx</td>
<td><a href="https://tukaani.org/xx/">https://tukaani.org/xx/</a></td>
<td>-9</td>
</tr>
<tr>
<td>zstd</td>
<td><a href="http://facebook.github.io/zstd/">http://facebook.github.io/zstd/</a></td>
<td>-19</td>
</tr>
<tr>
<td>bzip2</td>
<td><a href="https://sourceware.org/bzip2/">https://sourceware.org/bzip2/</a></td>
<td>-9</td>
</tr>
<tr>
<td>bsc</td>
<td><a href="http://libbsc.com/">http://libbsc.com/</a></td>
<td>-b2047</td>
</tr>
</tbody>
</table>

Table 2 Selected statistics on the inputs used in the experiments: $n$ is the file size, $\sigma = |\Sigma|$ the number of distinct characters (bytes), $H_0 := \sum_{i=1}^{\sigma}(n_i/n) \log_2(n_i/n)$ the zeroth-order entropy, $z_{78}$ the number of LZ78 phrases and $z_{77}$ the number of LZ77 phrases.

<table>
<thead>
<tr>
<th>Input</th>
<th>$n$</th>
<th>$\sigma$</th>
<th>$H_0$</th>
<th>$z_{78}$</th>
<th>$z_{77}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMONCRAWL</td>
<td>100 · $2^{20}$</td>
<td>243</td>
<td>6.20</td>
<td>7,149,629,111</td>
<td>2,596,068,363</td>
</tr>
<tr>
<td>DNA</td>
<td>100 · $2^{30}$</td>
<td>4</td>
<td>2.00</td>
<td>5,895,129,082</td>
<td>4,414,716,848</td>
</tr>
<tr>
<td>WIKIPEDIA</td>
<td>100 · $2^{30}$</td>
<td>213</td>
<td>5.37</td>
<td>5,340,147,122</td>
<td>3,513,405,107</td>
</tr>
</tbody>
</table>

we encode the length 255 followed by the binary representation of $\ell - 255$ using $\lceil \log B \rceil$ bits. If the phrase is a literal phrase ($\ell = 0$), $\delta \in \Sigma$ is a character that we encode using a Huffman code as for topk-lz78. If the phrase is a top-$k$ phrase ($\ell = 1$), we encode the top-$k$ trie node number $\delta < k$ in binary using $\lceil \log (k - 1) \rceil$ bits. Otherwise, if the phrase is referencing ($\ell > 1$), then we encode $\delta$ in binary using at most $\lceil \log \delta_{\max} \rceil$ bits (where $\delta_{\max} < B$ is the largest $\delta$ encountered among the $b$ tuples).

In blockwise-lz77, tuples are also of the form $(\ell, \delta)$, but there are only literal or referencing phrases and no top-$k$ phrases.

6.2 Experimental Setup

The experiment is conducted on a machine with an Intel Xeon E5-2640v4 processor running at 2.4 GHz with a 25 MB cache and 64 GB of RAM. We measure the average compression time over three iterations as well as the compression ratio (output size divided by input size) for our compressors from section 6.1 and the compressors listed in table 1, which we execute with flags for lowest compression ratio. For a fair comparison, we disabled the parallel computation features of bsc by compiling it without OpenMP. Our code is compiled with GCC 13.2 and all optimizations enabled (-O3). We set $k$ as high as possibly in our compressors to fill the available RAM. Preliminary experiments revealed that the threshold $\theta$ increases only very rarely for large $k$, confirming our assessments of section 3. This allows us to set $f_{\max}'$ very low with no need for any renormalization. Specifically,

- for topk-lz78, we set $k := 896 \cdot 2^{20}$ and $f_{\max}' := 2^{20}$,
- for topk-lz77, we set $B := 2^{31} - 1$, which is the largest block size that allows us to use 32-bit indices for computing the LZ77 parsing. We set $k := 640 \cdot 2^{20}$ and $f_{\max}' := 2^{20}$, and
- for blockwise-lz77, we set $B := 2^{31} - 1$ for a direct comparison with topk-lz77.
- for blockwise-lz77+, we set $B' := 3 \cdot 2^{30} \approx 1.5B$ to fill the available RAM (here, 64-bit indices are required to randomly access the block).

The input files for the compressors are COMMONCRAWL (web crawls from the Common Crawl Project, text only), DNA (raw human genomic sequences consisting of only A, C, G and T) and WIKIPEDIA (XML dumps of the German and English versions of Wikipedia). Each file is of size 100 GiB. Table 2 shows additional statistics on the inputs. In appendix B, we give more detailed information as to how they have been generated.
6.3 Results

Our main results are shown in figure 4, where we plot the running times and achieved compression ratios of the stated compressors for the individual inputs.

Among the LZ-based compressors (gzip, xz, zstd and ours), our top-k LZ77 approximation topk-lz77 achieves the best combination of compression time and ratio. Even though xz can achieve slightly lower compression ratios (under 1% lower on DNA and Wikipedia, and nearly five times as long on DNA). The running times are also roughly independent of the input with a standard deviation of only approximately 37 minutes. This is in contrast to the other LZ-based compressors, which take substantially longer to compress DNA (for computing the next phrase, gzip generates candidate lists based on the first three characters and then processes these rather naively – if the alphabet is small and near-uniformly distributed, the lists become very long; xz and zstd presumably use similar heuristics).

As expected – due to the fact that LZ78 typically produces a larger number of phrases than LZ77 – the compression ratios of our LZ78 approximation topk-lz78, albeit comparatively fast, are among the worst. However, we note that computing the exact LZ78 factorization of the inputs could not be computed within the available memory.

The block-sorting compressor bsc achieves the best compression ratios by far, which indicates that compression based on the Burrows-Wheeler Transform (BWT) works very well for natural languages. Since it involves mainly suffix sorting – a problem very well studied in practice – it achieves this also at relatively short running times. Even though bzip2 is based on the same fundamentals, it uses much smaller block sizes and thus achieves only much higher compression ratios, albeit also very fast.

Impact of $k$

To analyze the impact of the parameter $k$ on compression and resource usage, we run topk-lz77 and topk-lz78 for different $k$ and present the results in figures 5 (appendix A).

The memory usage is approximately 60 B per trie node. The block text index used by topk-lz77 requires 8 B per character (suffix array, its inverse, and the LCP array). Therefore, with the chosen $B := 2^{31} - 1$, topk-lz77 requires almost 16 GiB of memory additional to
the top-k trie and independent of k. Despite the fact that our algorithms require constant (amortized) time per input character, the running times become longer for larger k. This can be explained by the higher number of cache misses that occur navigating the trie and updating the Space-Saving data structure: for the largest k, these are spread across the majority of the available RAM and it becomes far more likely that every navigation step (either in the top-k trie or the Space-Saving data structure) incurs a cache miss.

The parameter k affects the number of phrases emitted by topk-lz78 significantly. As an example, CommonCrawl has 7,149,629,111 LZ78 phrases. For k = 2^{20}, topk-lz78 emits 15,661,924,249 (more than twice as many), but only 7,910,221,374 phrases (10.6% more than LZ78) for the largest k = 896 · 2^{20}. The compression ratio improves accordingly. The difference between the number of emitted phrases and the number z_{78} of actual LZ78 phrases increases only sublinearly with decreasing k for all inputs. Empirically, this respects our conjecture of section 4 that topk-lz78 is a \( \Theta(\sqrt{n}/k) \) approximation of LZ78.

For topk-lz77, there is a much less visible effect. We refer to figure 6 for a closer look on the number of phrases. For k = 2^{20}, on average, topk-lz77 emits only less than 0.5% fewer phrases than blockwise-lz77 and can hardly be considered an improvement. For the largest k = 2^{20}, it emits approximately 2.5% phrases less than blockwise-lz77 on average. Increasing k helps only marginally to approach the number z_{77} of actual LZ77 phrases.

**Top-k LZ77 versus (blockwise) LZ77**

To give a more detailed comparison, table 3 (in appendix A lists the number of phrases emitted by blockwise-lz77 and topk-lz77 (for k = 640 · 2^{20}) in comparison to the number z_{77} of LZ77 phrases. The top-k trie contributes significantly to the output: on average, 55.6% of the phrases are top-k phrases (70.9% on CommonCrawl, 36.6% on DNA and 59.2% on Wikipedia). This indicates that the trie contains frequent patterns typically longer than referencing (block-local) LZ phrases starting at the same position.

As stated previously, each trie node occupies approximately 60 B of memory, whereas each additional character in the block only adds 8 B to the text index size. Even though these are not directly comparable, a fair question is why not simply increase the block size to fill the RAM instead of using the top-k trie? To address this, consider blockwise-lz77+, where we set the block size accordingly (1 GiB larger than blockwise-lz77+). An important difference is that now, with blocks \( \geq 2^{31} \), each character contributes 16 B to the text index as we require 64 bits for block access. (We did not consider implementing, e.g., 40-bit integers). In table 3, we can see that indeed, blockwise-lz77+ emits slightly less phrases than topk-lz77 (4.0% less on CommonCrawl, 1.9% less on DNA and 0.9% less on Wikipedia). However, from figure 4, we can take that topk-lz77, albeit somewhat slower, still yields better compression ratios, which implies that top-k phrases can be encoded more efficiently.

**7 Conclusions and Outlook**

Our top-k LZ77 approximation achieves competitive compression ratios among other Lempel-Ziv-based compressors. This is remarkable considering the fact that our chosen encoding (described in section 6.1) is far less sophisticated than that of the competitors and likely allows for future improvements. Furthermore, this result is achieved at competitive speeds. Our top-k LZ78 approximation is a viable alternative to LZ78 if memory is limited, yielding good compression (albeit not competitive) at relatively fast running times.

To improve running times, parallel variants of our compressors could be considered. The key issue would be how to use the top-k trie, which currently only supports strictly sequential operations. If p is the number of processors, working with p independent tries that are
synchronized regularly can be an option. Then, however, each trie can only be of size at most \( \mathcal{O}(k/p) \) to stay within \( \mathcal{O}(k) \) total space. Synchronizing (e.g., merging) the \( p \) tries is also non-trivial, as the node numbering may be different in each of the tries.

To improve compression, an idea that we did not pursue in this work would be to encode more frequent patterns with shorter codewords than less frequent patterns. For example, instead of encoding simply the index \( v < k \) of a node in the trie using \( \lceil \log k \rceil \) bits, one could encode the tuple \( (f'_v, v') \), where \( f'_v \) is the frequency of the pattern represented by \( v \) and \( v' \) is \( v \)'s index within the bucket corresponding to frequency \( f'_v \). Because larger \( f'_v \) will naturally be encoded more frequently, they could be encoded using Huffman codes, much like the phrase lengths. The issue that arises with this particular proposal would be how to locate and access \( v' \) efficiently in dynamic buckets (currently, they are doubly-linked lists).

The top-\( k \) trie as presented tends to contain redundant information. This comes from the fact that every proper substring of a frequent pattern is also frequent. Particularly, consider a proper suffix \( v \) of a frequent pattern \( u \) that may also be frequent independently of \( u \) (not all occurrences of \( u \) imply an occurrence of \( v \)). Every pattern of which \( v \) is a proper prefix is contained in an additional branch off the trie’s root, potentially rephrasing many substrings also contained in the path to \( u \). To make most of the working space, an important open goal is therefore to compress the trie or use a different representation altogether. Ideally, based on the idea of string attractors [23], it could be of interest to maintain dynamically the smallest string that is a \( k \)-attractor for the input, where \( k \) is a parameter to control the space usage.

A more general issue with using heavy hitters for compression is that they tend to consist of shorter patterns, as these are naturally more frequent. Going away from a strict streaming model, future work may consider a precompression pass, where a different algorithm, aimed specifically at finding long repetitions, preprocesses the input.

It is straightforward to think about random access to the compressed string. In the presented encoding, a node index \( v \) always refers to the trie \( T_F \) in the instant in which it is was encoded. Accessing a particular character \( S[i] \) would therefore require to decode the entire prefix \( S[1..i] \). It may be interesting to consider a two-pass variant of top-\( k \) LZ78 similar to the work of Arz and Fischer [7]: in the first pass, we compute the top-\( k \) trie \( T_F \) for all of \( S \). In the second pass, we then compute a greedy parsing of \( S \) using the fixed trie \( T_F \). By keeping \( T_F \) in RAM and precomputing a suitable rank/select data structure for the phrases, it is then possible to access any \( S[i] \) in time at most \( \mathcal{O}(h) \), where \( h \) is the height of \( T_F \).

Looking past compression, the top-\( k \) trie may have applications as a framework for online frequent pattern estimation where frequent patterns are of interest, e.g., for mining key phrases in a document. In this work, we did not consider output of frequent patterns, but this can easily be done by traversing the trie after processing the input. Depending on the application, the error introduced by our approach as discussed in section 3 may be relevant.

References


A Additional Data

This appendix contains figures showing additional data from the experiments of section 6, namely:

- Figures 5 and 6 (impact of $k$ on the number of phrases, compression ratio, running time and required memory of topk-lz78 and topk-lz77
- Table 3 (comparison of the number of phrases between exact, blockwise and top-k LZ77)
Figure 5 Impact of $k$ on the number of emitted phrases, compression ratio, running time (average of three iterations) and memory usage of our algorithms. Regarding the number of phrases, the dotted line marks the number of phrases emitted by blockwise-lz77 (upper bound) and the dashed lines mark the exact number of LZ77 or LZ78 phrases, respectively (lower bounds). Please refer to figure 6 for a closer look at the number of phrases emitted by topk-lz77.

Figure 6 Impact of $k$ on the number of phrases emitted by topk-lz77. This shows the same data as figure 5, but on a different scale for each input. The dotted lines mark the number of phrases emitted by blockwise-lz77 (upper bound).
Table 3 Phrases emitted by blockwise-lz77, topk-lz77 \((k = 640 \cdot 2^{20})\) and blockwise-lz77+ compared to the number \(z_{77}\) of LZ77 phrases. Values are given as billions. For topk-lz77, we distinguish between the number of (block-local) LZ phrases as well as phrases from the top-\(k\) trie.

<table>
<thead>
<tr>
<th>Input</th>
<th>blockwise-lz77+</th>
<th>Total</th>
<th>5</th>
<th>4</th>
<th>(z_{77})</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMONCRAWL</td>
<td>5.554</td>
<td>1.502</td>
<td>4.052</td>
<td>5.333</td>
<td>2.596</td>
</tr>
<tr>
<td>DNA</td>
<td>5.351</td>
<td>3.178</td>
<td>2.173</td>
<td>5.251</td>
<td>4.415</td>
</tr>
<tr>
<td>WIKIPEDIA</td>
<td>4.563</td>
<td>1.797</td>
<td>2.765</td>
<td>4.520</td>
<td>3.513</td>
</tr>
</tbody>
</table>

B Input Corpus

For the experiments in section 6, we use the 100 GiB prefixes of the following corpus.

- **COMMONCRAWL**: Web crawls from the Common Crawl Project (https://commoncrawl.org). We use the WET files 00000 to 00600 from the CC-MAIN-2019-09 crawl and concatenate them in numerical order, removing all WARC meta information (one line consisting of WARC/1.0 and the following eight lines).

- **DNA**: Human genomic sequences from the European Nucleotide Archive (https://www.ebi.ac.uk). We extract the raw sequences from the FASTQ datasets DRR00000 to DRR000426 (non-continuous) available at ftp://ftp.sra.ebi.ac.uk/vol1/fastq/DRR000, removing all characters other than \(A, C, G\) and \(T\). These are concatenated in numerical order (DR<\#>, where <\#> is the file number).

- **WIKIPEDIA**: XML dumps of the German and English versions of Wikipedia (dumps.wikimedia.org). We concatenate the dumps as of March 20, 2019 in the order de, en. The contained articles are of that date with no version history.

C Computing LZ77 in Linear Time and Space

To compute a LZ77 parsing of a string \(S \in \Sigma^n\) of length \(n\) – as required by our blockwise LZ77 implementations (with and without the top-\(k\) trie) – we implement a well-known approach that simulates the longest previous factor (LPF) array as shown in algorithm 2.

We first compute the suffix array \(A\) of \(S\) and its inverse \(A^{-1}\), which can be done in time and space \(O(n)\) (see, e.g., [22]). With this, we parse \(S\) from left to right computing the next LZ77 phrase in each step.

Let \(i < n\) be the current position in \(S\). The next LZ77 phrase is the longest prefix of \(S[i..n]\) that has an occurrence in \(S\) beginning at a position \(j < i\). We can find \(j\) in \(i\)'s *neighbourhood* in the suffix array. Let \(i' = A^{-1}[i]\) be the position of \(i\) in the suffix array. Because of the lexicographic order of suffixes in \(A\), the values nearest to \(i'\) in \(A\) that are \(< i\) are candidates for \(j\). We call them the *previous smaller value* (PSV, lexicographically smaller than \(S[i..n]\)) or the *next smaller value* \(j_2\) (NSV, lexicographically larger than \(S[i..n]\)) and denote them by \(j_1\) or \(j_2\), respectively. By computing the *longest common extension* (LCE) between \(S[i..n]\) and \(S[j_1..n]\) or \(S[j_2..n]\), respectively, we find the lengths \(\ell_1\) and \(\ell_2\) of the candidate LZ77 phrases. Because LZ77 phrases are defined greedily, whichever candidate is longer is the next LZ77 phrase. Ties are broken arbitrarily. The special case where \(\ell_1 = \ell_2 = 0\) occurs if \(S[i]\) is the first occurrence of the corresponding character in \(S\), then the next LZ77 phase is a literal phrase. The algorithm requires time and space \(O(n)\) (the LCE computations can be done using the LCP array, which can be computed in \(O(n)\) extra time and space, or naively comparing character by character, contributing to at most two additional scans of \(S\) in total). For a more detailed elaboration of this algorithm, we refer the reader to [20].
Algorithm 2  Computing a LZ77 parsing for an input $S$ by simulating the LPF array.

For clarity, we omitted the handling of some border cases in the LCE computations. The operator $\circ$ concatenates two strings.

\begin{algorithm}
\begin{align*}
\textbf{Input} & : \text{String } S \in \Sigma^n \text{ of length } n \\
\textbf{Output} & : \text{LZ77 parsing } S' \text{ of } S \\
1 & S' \leftarrow \epsilon \\
2 & A \leftarrow \text{suffix array of } S, A^{-1} \leftarrow \text{inverse suffix array of } S \\
3 & i \leftarrow 1 \\
4 & \textbf{while } i < n \textbf{ do} \\
5 & \quad i' \leftarrow A^{-1}[i] \quad // \text{suffix array neighbourhood of } i \text{ in } A \\
6 & \quad j_1 \leftarrow A[\max\{j' \mid j' < i' \land A[j'] < i]\} \quad // \text{PSV in } A \text{ w.r.t. } i' \\
7 & \quad \ell_1 \leftarrow \max\{\ell \mid S[i .. i + \ell] = S[j_1 .. j_1 + \ell] \land S[i + \ell + 1] \neq S[j_1 + \ell + 1]\} \quad // \text{LCE} \\
8 & \quad j_2 \leftarrow A[\min\{j' \mid j' > i' \land A[j'] < i]\} \quad // \text{NSV in } A \text{ w.r.t. } i' \\
9 & \quad \ell_2 \leftarrow \max\{\ell \mid S[i .. i + \ell] = S[j_2 .. j_2 + \ell] \land S[i + \ell + 1] \neq S[j_2 + \ell + 1]\} \quad // \text{LCE} \\
10 & \quad \ell \leftarrow \max\{\ell_1, \ell_2, 1\} \\
11 & \quad \textbf{if } \ell > 1 \textbf{ then} \\
12 & \quad \quad \delta \leftarrow \begin{cases} i - j_1 & \text{if } \ell_1 > \ell_2 \\
13 & i - j_2 \quad \text{otherwise} \\
14 & \end{cases} \\
15 & \quad S' \leftarrow S' \circ (i, \delta) \quad // \text{referencing phrase} \\
16 & \quad i' \leftarrow i + \ell \quad \text{else} \\
17 & \quad S' \leftarrow S' \circ (0, S[i]) \quad // \text{literal phrase}
\end{align*}
\end{algorithm}

In our implementation (algorithm 2), we implicitly convert referencing phrases of length one to literal phrases (line 10), which is more beneficial towards the encoding. We do not use the LCP array but compute LCEs naïvely by scanning. This requires significantly less working memory and is, in practice, less time-consuming than computing the LCE array and accessing it randomly. We use libsais for computing the suffix array and inverse it trivially in $n$ steps ($A^{-1}[A[i]] := i$ for all $i$). Thus, if $n < 2^{31}$, we can use 32-bit integers and the practical space requirement, including the input $S$, becomes 9n bytes (libsais requires the 32nd bit to temporarily store auxiliary information). For larger $n$, we use 64-bit integers, such that the requirement becomes $17n$ bytes. (It would be possible to implement custom integer types such as 40-bit integers, but we did not consider that for this work.)

C.1 Fast Semi-External Memory Implementation

For computing the number $z = |S'|$ of LZ77 phrases of the 100 GiB inputs of section 6, we faced a practical problem: the machine with the largest RAM available to us has 1 TB, which is slightly less than 10n bytes. The algorithm described above, however, requires 17n bytes (since $n \geq 2^{31}$, we require 64-bit integers).

We considered using succinct representations of the suffix array (e.g., [33] as provided by the SDSL), however, the working space required for their construction exceed our limitations. We furthermore considered using the external-memory algorithms by Kärkkäinen et al. [21], however, these would have required several weeks of running time per input file.

To tackle this, we implemented a semi-external memory algorithm that requires 9n bytes of RAM for 64-bit integers and 8n bytes of external memory (16n during construction) and is included in our public source code (1pfs). It is an adaptation of algorithm 2 that works as follows:
1. Load $S$ and compute $A$ as usual using $9n$ bytes of RAM.
2. Write $A$ to file $sa$.
3. Compute $A^{-1}$ in the space allocated for $A$ by streaming the file $sa$. Let $x$ be the $i$-th value read from the file, then we set $A^{-1}[x] := i$.
4. Write $A^{-1}$ to file $isa$.
5. Load file $sa$ back into $A$ and delete file $sa$.
6. To parse $S$, stream the values $A^{-1}[i]$ from file $isa$ and skip entries as needed.
7. When done parsing $S$, delete file $isa$.

The external memory portions of the algorithm are optimal in the external memory model in the sense that they require only $O(n/B)$ I/O operations.

Since $n < 2^{40}$ in our case, we encoded the entries of files $sa$ and $isa$ using 40 bits each. Thus, the external memory requirement dropped to $5n$ bytes (10n bytes during construction). We used this algorithm to successfully compute the LZ77 parsings for the stated inputs within approximately 24 hours each.
Taxonomic Classification with Maximal Exact Matches in KATKA Kernels and Minimizer Digests

Dominika Draesslerová
Czech Technical University in Prague, Czech Republic

Omar Ahmed
Johns Hopkins University, Baltimore, MD, USA

Travis Gagie
CeBiB & Dalhousie University, Halifax, Canada

Jan Holub
Czech Technical University in Prague, Czech Republic

Ben Langmead
Johns Hopkins University, Baltimore, MD, USA

Giovanni Manzini
University of Pisa, Italy

Gonzalo Navarro
CeBiB & DCC, University of Chile, Chile

Abstract
For taxonomic classification, we are asked to index the genomes in a phylogenetic tree such that later, given a DNA read, we can quickly choose a small subtree likely to contain the genome from which that read was drawn. Although popular classifiers such as Kraken use \(k\)-mers, recent research indicates that using maximal exact matches (MEMs) can lead to better classifications. For example, we can

- build an augmented FM-index over the the genomes in the tree concatenated in left-to-right order;
- for each MEM in a read, find the interval in the suffix array containing the starting positions of that MEM’s occurrences in those genomes;
- find the minimum and maximum values stored in that interval;
- take the lowest common ancestor (LCA) of the genomes containing the characters at those positions.

This solution is practical, however, only when the total size of the genomes in the tree is fairly small. In this paper we consider applying the same solution to three lossily compressed representations of the genomes’ concatenation:

- a KATKA kernel, which discards characters that are not in the first or last occurrence of any \(k_{\text{max}}\)-tuple, for a parameter \(k_{\text{max}}\);
- a minimizer digest;
- a KATKA kernel of a minimizer digest.

With a test dataset and these three representations of it, simulated reads and various parameter settings, we checked how many reads’ longest MEMs occurred only in the sequences from which those reads were generated (“true positive” reads). For some parameter settings we achieved significant compression while only slightly decreasing the true-positive rate.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Taxonomic classification, metagenomics, KATKA, maximal exact matches, string kernels, minimizer digests
10:2  Taxonomic Classification with MEMs

Supplementary Material Software (Source Code): https://github.com/draessld/KATKA2
archived at swh:1:dir:8b5cba2a90c1418c7a467f89d0db4ef7b6c938f

Funding  Omar Ahmed and Ben Langmead: funded by NIH grants R35GM139602 & R01HG011392
and NSF grant DBI-2029552.
Travis Gagie and Gonzalo Navarro: partially funded by Basal Funds FB0001, ANID, Chile.
Dominika Draesslerová: funded by CTU grant No. SGS23/205/OH/3/T/18 and by ROBOPROX
reg. no. CZ.02.01.01/00/22_008/0004590.
Travis Gagie: funded by NSERC Discovery Grant RGPIN-07185-2020.
Giovanni Manzini: funded by the Italian Ministry of Health, POS 2014-2020, project ID T4-AN-
07, CUP I53C22001300001, by INdAM-GNCS Project CUP E53C23001670001 and by PNRR
ECS00000017 Tuscany Health Ecosystem, Spoke 6 CUP I53C22000780001.

Acknowledgements Many thanks to Sana Kashgouli and Finlay Maguire for helpful discussions.

1  Introduction

Kraken [28] is probably the best-known metagenomic tool for taxonomic classification. Given
a phylogenetic tree for a collection of genomes and a value $k$, it stores an index mapping
each $k$-mer in the collection to the root of the lowest subtree containing all occurrences of
that $k$-mer. Later, given a DNA read – which may not match exactly in any of genomes in
the collection – it tries to map all the $k$-mers in that read to subtrees in the tree and then to
choose a small subtree likely to contain the source of the read. For example, if Kraken is
given the toy phylogenetic tree shown at the top of Figure 1 and $k = 3$, then it will store the
$k$-mer index shown at the bottom of that figure. Later, given the toy read ATAC, it will map
ATA to 6 and TAC to 2. Since the subtree rooted at 6 contains the one rooted at 2, it will
report that the read probably came from a genome in the subtree rooted at 2.

Nasko et al. [18] showed that a static choice of $k$ is problematic, since “the [reference]
database composition strongly influence[s] the performance”, with larger $k$ values working
better as the collection of genomes grows over time. Limiting all analyses to a single choice of
$k$ causes other problems as well. First, some branches of the taxonomic tree are well studied
and contain a large number of genome assemblies for diverse strains and species. Other
branches are scientifically significant but harder to study, and contain only a few genomes.
In the more richly sampled spaces, larger values of $k$ will better allow for discrimination at
deeper levels of the tree.

Choosing a constant value for $k$ also conflicts with the varying error rates across sequencing
technologies. For the high-accuracy Illumina technology, we expect longer matches to the data
base and should favour a larger $k$. For a high-error-rate technology like Oxford Nanopore, we
expect shorter matches and a small $k$ is better. To this end, many widely tools for classifying
long (error-prone) reads use matching statistics and/or full-text indexes [15, 1], as do some
for short reads [14, 17]. Nasko et al. observed that

“alternative approaches to traditional $k$-mer-based [lowest common ancestor] identifi-
cation methods, such as those featured within KrakenHLL [4], Kallisto [3], and
DUDes [21], will be required to maximize the benefit of longer reads coupled with
ever-increasing reference sequence databases and improve sequence classification
accuracy.”

Cheng et al. [6] showed that finding the maximal exact matches (MEMs) of the read
with respect to the collection and then mapping each MEM to the root of the lowest subtree
containing all occurrences of that MEM, gives better results than mapping $k$-mers for any
single $k$. However, they did not give a space- and time-efficient index for finding and mapping MEMs. As a potential step toward working with MEMs, Gagie et al. [10] described an LZ77-based index KATKA that takes $O(z \log n)$ space, where $z$ is the number of phrases in the LZ77 parse of the collection of genomes and $n$ is the total length of the collection, and works like Kraken but taking $k$ at query time instead of at construction time.

KATKA finds the indices of the genomes containing the first and last occurrences of each $k$-mer in the collection, then performs a lowest common ancestor (LCA) query on those genomes in the tree to find the root of the smallest subtree containing all the occurrences of that $k$-mer. As far as we know, however, there is no practical way to find MEMs with LZ77- or grammar-based indexes, even if there have been some promising recent developments [12, 19] in this direction. Thus, KATKA is not yet a practical implementation of Cheng et al.’s idea.

Since an LCA data structure for the phylogenetic tree takes a constant number of bits per genome, the main challenge to implementing Cheng et al.’s idea is to find the MEMs of the read with respect to the collection and then to find the genomes containing the first and last occurrence of each MEM. We call all this information the MEM table for the read. We describe in Section 2 how we can extend a technique by Ohlebusch et al. [20] to build the MEM table in constant time per character in the read plus $O(\log n)$ time per MEM as long as we are willing to use an $O(n)$-bit augmented FM-index for the collection – but a space usage of $O(n)$ bits is prohibitive when the collection is large and anyway wasteful when it is highly repetitive. The most practical way we know of to build the MEM table is with Cáceres and Navarro’s [5] block-tree compressed suffix tree, but that offers more functionality than we need at the cost of using more space than we would like (“1–3 bits per symbol in highly repetitive text collections”).

In this paper we build approximations of MEM tables using augmented FM-indexes over
- a string kernel for the collection,
- a minimizer digest for the collection,
- a string kernel for a minimizer digest for the collection.
String kernels and minimizer digests are lossily compressed representations of strings, which we review in Section 2. We need a special kind of string kernel that we call a KATKA kernel and define also in Section 2. We can use KATKA kernels and minimizer digests to reduce the size of the augmented FM-index, at the cost of limiting the lengths of matches and reporting some false-positive matches. To test how we can trade off accuracy for compression, we built augmented FM-indexes over a test dataset and KATKA kernels, minimizer digests, and KATKA kernels of minimizer digests for that dataset with various parameter settings, and checked for how many of a set of simulated reads their longest MEMs occurred only in the sequences from which those reads were generated (“true positive” reads). For some parameter settings we achieved significant compression while only slightly decreasing the true-positive rate.

2 Preliminaries

2.1 Augmented FM-indexes

Ohlebusch et al. [20] showed how, if we store an augmented FM-index, then given a read we can find its MEMs quickly. We first show how to extend their technique to computing the MEM table in constant time per character in the read and $O(\log n)$ time per MEM.

Suppose each genome in the collection is terminated by a special separator character $\$\$ as shown in Figure 2. The augmented FM-index consists of data structures supporting access, rank and select on the collection’s Burrows-Wheeler Transform (BWT)\(^1\); access, range-minimum and range-maximum on their suffix array (SA); range-minimum, range-maximum, previous smaller value (PSV) and next smaller value (NSV) queries on their longest common prefix (LCP) array; and rank on the bitvector $B$ with a 1 marking each $\$\$ in the collection.

As long as the collection is over a constant-size alphabet, these data structures together take $O(n)$ bits with all their queries taking at most $O(\log n)$ time. They are also implemented in the Succinct Data Structure Library (SDSL) [13] as components of a compressed suffix tree.

Given the read $\text{ACATA}$, for example, we start a backward search with BWT interval $\text{BWT}[0..44]$ (the entire BWT). After 3 backward steps we find the interval $\text{BWT}[16..18]$ for $\text{ATA}$. Since this interval does not contain a copy of the preceding character $\mathcal{C}$ in the read, we know $\text{ATA}$ is a MEM of $\text{ACATA}$ with respect to the collection. We use range-minimum and range-maximum queries over $\text{SA}[16..18]$ and access to SA to determine that the first and last occurrences of $\text{ATA}$ start at positions 11 and 41 in the collection. Since $B.\text{rank}_1(11) = 1$ and $B.\text{rank}_1(41) = 4$, we know those occurrences are in the second and fifth genomes in the collection (stored at nodes 3 and 9 in the phylogenetic tree). Notice we consider only the first and last occurrences and not the occurrence starting at position 19, for example.

We then use rank and select queries on the BWT to look for the previous copy $\text{BWT}[14..16]$ of $\mathcal{C}$ and next copy of $\mathcal{C}$ (which does not exist); use a range-minimum query on $\text{LCP}[14+1 = 15..16]$ to find the position 16 of the length 2 of the longest prefix $\text{AT}$ of $\text{ATA}$ that is preceded by $\mathcal{C}$ in the collection; use access to the LCP to retrieve that value 2; and use $\text{PSV}(16) = 12$ and $\text{NSV}(16) = 22$ queries to find the interval $\text{BWT}[12..22] = 21$ for that prefix $\text{AT}$. After 2 backward steps we find the interval $\text{BWT}[6..8]$ for $\text{ACAT}$. We use range-minimum and range-maximum queries over $\text{SA}[6..8]$ and access to SA to determine that the first and last occurrences of $\text{ACAT}$ start at positions 4 and 21 in the collection. Since $B.\text{rank}_1(4) = 0$ and $B.\text{rank}_1(21) = 2$, we know those occurrences are in the first and third genomes in the collection (stored at nodes 1 and 5 in the phylogenetic tree).

---

\(^1\) To reduce the size of the figure we have actually shown the genomes’ extended BWT [16], which is functionally equivalent as far as we are concerned as long as each genome has length $\Omega(\log n)$. Notice some LCP values, such as LCP[4], “wrap around” and count a character in the BWT.
2.2 String kernels and KATKA kernels

Ferrada, Gagie, Hirvola and Puglisi [8, 11] and Prochazka and Holub [22] (see also [9]) independently defined the order-k_max kernel of a string to be the subsequence consisting of the characters in the first occurrence of any distinct k_max-mer in the string, with maximal omitted substrings replaced by copies of a new separator character $. Since we want to find the first and last occurrences of matches, we define the order-k_max KATKA kernel of a collection of genomes essentially the same way, but with the subsequence consisting of the characters in the first or last occurrence of any distinct k_max-mer in the string, and the copies of the separator character $. Since reads will not contain $, we also do not replace with $ all maximal omitted substrings adjacent to copies of $.

By construction, for k \leq k_max, every k-mer from the normal alphabet (so not including $) in the original string occurs in the KATKA kernel and vice versa. Moreover, if there are i copies of $ to the left of the first occurrence of such a k-mer in the kernel, then the first occurrence of that k-mer in the collection is in the (i + 1)st genome (and symmetrically for the last occurrences). The running example we have used so far is too small to illustrate properly the advantages and disadvantages of KATKA kernels, so Figure 3 shows a slightly
larger collection of slightly longer toy genomes and Figure 4 shows the subsequence consisting of the characters in the first or last occurrence of each distinct 4-mer and the copies of $. Figure 5 shows the 4th-order KATKA kernel of the collection with maximal omitted substrings replaced by copies of #. In this example, the 4th-order KATKA kernel is about half the size of the original collection, but this varies in practice depending on $k_{\text{max}}$ and the size and repetitiveness of the collection. The 5th-order KATKA kernel, which we do not show, is about 70% of the size of the original collection.

2.3 Minimizer digests

To build a minimizer digest [24] for a string $S[1..n]$, we
1. choose parameters $k$ and $w$ and a hash function $h(\cdot)$ function on $k$-mers,
2. mark each $k$-mer $S[j..j+k-1]$ in $S$ such that $h(S[j..j+k-1])$ is the leftmost occurrence of the minimum in $h(S[i..i+k-1], \ldots, S[i+w-1..(i+w-1+k-1)])$ for some $i$ with $i \leq j < i + w$,
3. return the sequence of marked $k$-mers’ hashes.

For example, suppose $k = 3$, $w = 10$ and the hash function $h(\cdot)$ takes a triple over $\{A, C, G, T\}$ as a 3-digit number $x$ in base 4 and returns $(2544x + 3937) \mod 8863$. The minimizer digests for the toy genomes in Figure 3 (excluding $\$$) are shown in Figure 6 separated by $\$$ and with the 64 triples over $\{A, C, G, T\}$ mapped to ASCII values between 37 and 100. Minimizer digests are widely used in bioinformatics to reduce tools’ time and space requirements; for example, they are used this way in Kraken 2 [27], mdBG [7] and SPUMONI 2 [2].

We note that although the first minimizer digest $=c\text{<J_cA\2X<\G2\cK\NJX5}$ is 21 characters while the first genome is 100 characters, the digest is over an alphabet of size 64 instead of 4; therefore, the minimizer is 126 bits while the genome is 200 bits. The space of the auxiliary data structures for an augmented FM-index for the minimizer digest still depends on the number 21 of characters in the digest, however.

We say the concatenation of the minimizer digests for the genomes in a collection, separated by $\$$, is the minimizer digest for the collection. By construction, if $\alpha$ is the minimizer digest for a pattern and there are $i$ copies of $\$$ to the left of the first occurrence of $\alpha$ in the minimizer digest for the collection, then the first occurrence of the pattern cannot be before the $(i + 1)$st genome (and symmetrically for the last occurrences) – although the pattern may not occur in that genome and possibly not in the whole collection.

2.4 KATKA kernels of minimizer digests

Of course, we can also build KATKA kernels of minimizer digests. Figure 7 shows the subsequence consisting of the characters in the first or last occurrence of each distinct pair – the 2nd-order KATKA kernel – and the copies of $\$$ in Figure 6, with maximal omitted substrings replaced by copies of #. It consists of 220 6-bit characters (1320 bits) plus the 16 $\$$s; the original minimizer digest consists of 287 6-bit characters (1722 bits) plus the $\$$s, the 4th-order KATKA kernel consists of 798 2-bit characters (1596 bits) plus the $\$$s, and the collection of toy genomes itself consists of 1600 2-bit characters (3200 bits) plus the $\$$s. We note that pairs of minimizers with $k = 3$ and $w = 10$ can represent substrings as short as 4 characters or as long as 17 characters in the genomes; in our example, on average a pair of minimizers represents about $2 \cdot (1600/287) \approx 11.15$ characters.

KATKA kernels of minimizer digests may inherit the strengths of both: with kernelization we can take advantage of repetition to compress, while using minimizers allows us to keep the parameter $k$ in the kernelization small while still dealing with reasonably long patterns.
Figure 3 A slightly larger collection of slightly longer toy genomes.

Figure 4 The subsequence consisting of the characters in the first or last occurrence of each distinct 4-mer and the copies of $\#$, with omitted characters replaced by spaces.

Figure 5 The subsequence consisting of the characters in the first or last occurrence of each distinct 4-mer – the 4th-order KATKA kernel – and the copies of $\#$, with maximal omitted substrings replaced by copies of $\#$, except for those adjacent to $\#$.
10:8  Taxonomic Classification with MEMs

Figure 6  Minimizer digests for the toy genomes in Figure 3, separated by $s$.

Figure 7  The subsequence consisting of the characters in the first or last occurrence of each distinct pair – the 2nd-order KATKA kernel – and the copies of $s$ in Figure 6, with maximal omitted substrings replaced by copies of $\#$.

3  Approximating MEM tables with FM-indexes of KATKA kernels and minimizer digests

Once we have built a KATKA kernel or minimizer digest for a collection of genomes, or a KATKA kernel of a minimizer digest, we can build an augmented FM-index over it. For example, Figure 8 shows the first and last lines of the augmented FM-indexes for the 4th-order KATKA kernel in Figure 5; the minimizer digest in Figure 6; and the 2nd-order KATKA kernel of the minimizer digest, from Figure 7. In all three cases, we include an implicit end-of-file character less than any other.

Consider the pattern $P = \text{GGATGGGCTAGACGATCTTCTG}$, which we obtained by choosing the substring $\text{GGGTGGGTTAGACGATCTTCTG}$ of toy genome 9 in Figure 3 (numbering the genomes from 0) and changing two characters. The MEM table of $P$ with respect to all the toy genomes is shown on the left in Figure 9. The MEM table of $P$ with respect to the 4th-order KATKA kernel with $s$ and $\#$ shown in Figure 5, is shown in the center of Figure 9. (The MEM table of $P$ with respect to the 5th-order KATKA kernel is the same as its MEM table with respect to the genomes.) The minimizer digest of $P$ with $w = 10$ is $Q$, and the MEM table of that with respect to the minimizer digest of the collection is shown on the right of Figure 9; the MEM table with respect to the 2nd-order KATKA kernel of the minimizer digest is the same as the MEM table with respect to the minimizer digest.

Since $P$ comes from toy genome 9, following Wood, Lu and Langmead's [27] terminology in their presentation of Kraken 2, we classify MEMs’ [first, last] ranges as true positives if they are exactly [9,9], false positives if they exclude 9 but are not empty, vague positives if they include 9 and at least one other number, and false negatives if they are empty. The classification of the MEMs’ ranges in Figure 9 are shown below:

<table>
<thead>
<tr>
<th>true positives</th>
<th>false positives</th>
<th>vague positives</th>
<th>false negatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>[9]</td>
<td>[0, 1], [8], [11]</td>
<td>[0, 15], [4, 11]</td>
<td></td>
</tr>
<tr>
<td>[12, 14], [13], [15]</td>
<td></td>
<td>[6, 15], [8, 15]</td>
<td></td>
</tr>
</tbody>
</table>

Notice the ranges for MEMs with respect to the toy genomes and the 4th-order KATKA kernel can never be empty (assuming every distinct character in $P$ occurs in the genomes at least once), so those ranges cannot be false negatives. On the other hand, if we generate $P$ by changing characters in a way that disrupts every previous minimizer and creates new minimizers that are not in the last minimizer digest of the genomes, then we can get MEMs with respect to the minimizer digest or to the 2nd-order KATKA kernel of the minimizer digest, whose ranges are empty.
Figure 8 The first and last lines of the augmented FM-indexes for the KATKA kernel in Figure 5 (top) and the minimizer digest in Figure 6 (bottom).
10:10 Taxonomic Classification with MEMs

<table>
<thead>
<tr>
<th>MEM</th>
<th>first</th>
<th>last</th>
</tr>
</thead>
<tbody>
<tr>
<td>GGGATGGGCTAG</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>TAGACGATCTTCTGT</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>TGTG</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MEM</th>
<th>first</th>
<th>last</th>
</tr>
</thead>
<tbody>
<tr>
<td>GGATGGG</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>GGGC</td>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td>GGCT</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>GCTAG</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>TAGA</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>AGACG</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>GACGATC</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>ATCTTCT</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>TGTG</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>TGTG</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MEM</th>
<th>first</th>
<th>last</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>8</td>
<td>15</td>
</tr>
</tbody>
</table>

Figure 9 The MEM tables of \( P \) with respect to the toy genomes in Figure 3 (left), the 4th-order KATKA kernel in Figure 5 (center), and the minimizer digests in Figure 6 (right).

Looking at the MEM table of \( P \) with respect to the toy genomes, it is intuitive to give more weight to the longer MEM, which occurs only in genome 9. If on this basis we guess correctly that \( P \) came from genome 9, then we can consider \( P \) a true positive with respect to the toy genomes; unfortunately, the same is not true with respect to the 4th-order KATKA kernel, nor to the minimizer digest with \( w = 10 \).

4 Experiments

In order to present a concise comparison of results obtained with a full dataset with those obtained with KATKA kernels, minimizer digests, and KATKA kernels of minimizer digests, for this section we focus on true-positive rates rather than whole MEM tables. We classify a read as a true positive if its longest MEM is a true positive (or all its longest MEMs, in the case of a tie).

We wrote the code for our experiments (which computes full MEM tables) in C++ using SDSL [26] and posted it at https://github.com/draesld/KATKA2. We ran our experiments on a server at the Department of Computer Science of the Czech Technical University in Prague with 128 AMD EPYC 7742 64-Core CPUs and 504 GiB of RAM, running GNU/Linux Kernel 5.15.0.

We chose 1000 bacterial genera consecutive in the phylogenetic tree for 138.1 release of the SILVA SSU Ref NR99 database [23] of ribosomal RNA (rRNA) gene sequences. We concatenated the gene sequences for the genera, separated by $s$, and built augmented FM-indexes for that 167328343-character concatenation, and KATKA kernels, minimizer digests, and KATKA kernels of minimizer digests for it with various parameter settings:

- for KATKA kernels of the original concatenation, we used \( k = 5, 10, 15, 20, \ldots, 45, 50, 100 \);
- for minimizer digests, we used 3-mers as minimizers and set \( w = 5, 10, 15, 20, \ldots, 45, 50 \);
- for KATKA kernels of the minimizer digests, we used \( k = 5, 10, 15, 20, \ldots, 45, 50 \) and the same \( w \) values.

We included the kernel with \( k = 100 \) of the original concatenation to show that as \( k \) increases, the true-positive rate does approach the rate achieved with an index of the original concatenation.

For each genus \( g \), we simulated 500 reads of 200 base pairs each by choosing a random starting location in the reference sequence for \( g \) and mutating 1% percent of the bases uniformly across the read to simulate sequencing error. For each read and each index, we found all the read’s longest MEMs and checked whether all their [first, last] ranges contained only the ID of the reference sequence for \( g \). Figure 10 shows the index sizes and true-positive rates over all 500000 simulated reads, and Figure 11 shows an average search time per read.
Clearly, we can achieve significant compression while only slightly decreasing the true-positive rate and without paying a penalty in search time, especially with KATKA kernels of minimizer digests: for example, with $k = 30$ and $w = 5$ our index took 56.5 MiB and achieved a true-positive rate of 74.3%, compared to 287.9 MiB and 78.6% with an index for the full dataset; our index is also slightly faster than the index for the full dataset. This is a better compression/accuracy/speed tradeoff than we achieve with either kernelization or minimizers alone.

If we are willing to sacrifice the true-positive rate moderately, we can increase $w$ and also achieve significant speedups. FM-indexes over minimizer digests are known to be usually significantly faster than indexes over the original datasets, both because some characters are not represented in the digests and because we use a backward step for each minimizer rather than for each character, incurring fewer cache misses. Interestingly, however, we achieved slightly better speedups with both kernelization and minimizers than with either separately.

5 Conclusions and future work

Figure 10 strongly confirms our conjecture from Subsection 2.4 that KATKA kernels of minimizer digests can inherit the strengths of both. In the near future we plan to experiment also with varying the width of minimizers (for simplicity, in this paper we always used...
3-mers). Later, we plan to incorporate indexing KATKA kernels of minimizer digests to build MEM tables – with more sophisticated classifications that take advantage of all the information in those tables – into a full pipeline for taxonomic classification of reads.

The confirmation of our conjecture may be useful for other applications as well, when we are dealing with repetitive datasets and want the flexibility of an augmented FM-index (instead of an r-index or a grammar-based index, for example) but kernelization has still had less impact than we might have hoped, because setting the parameter $k$ high enough to allow for the pattern lengths used in practice results in poor compression. For example, an obvious question that arises from our work is whether Valenzuela et al.’s [25] PanVC tool can achieve interesting tradeoffs between compression and accuracy using kernelization of minimizer digests, instead of only kernelization.

References


Engineering Weighted Connectivity Augmentation Algorithms

Marcelo Fonseca Faraj
Heidelberg University, Heidelberg, Germany

Ernestine Großmann
Heidelberg University, Heidelberg, Germany

Felix Joos
Heidelberg University, Heidelberg, Germany

Thomas Möller
Heidelberg University, Heidelberg, Germany

Christian Schulz
Heidelberg University, Heidelberg, Germany

Abstract
Increasing the connectivity of a graph is a pivotal challenge in robust network design. The weighted connectivity augmentation problem is a common version of the problem that takes link costs into consideration. The problem is then to find a minimum cost subset of a given set of weighted links that increases the connectivity of a graph by one when the links are added to the edge set of the input instance. In this work, we give a first implementation of recently discovered better-than-2 approximations. Furthermore, we propose three new heuristics and one exact approach. These include a greedy algorithm considering link costs and the number of unique cuts covered, an approach based on minimum spanning trees and a local search algorithm that may improve a given solution by swapping links of paths. Our exact approach uses an ILP formulation with efficient cut enumeration as well as a fast initialization routine. We then perform an extensive experimental evaluation which shows that our algorithms are faster and yield the best solutions compared to the current state-of-the-art as well as the recently discovered better-than-2 approximation algorithms. Our novel local search algorithm can improve solution quality even further.

2012 ACM Subject Classification
Mathematics of computing → Optimization with randomized search heuristics; Theory of computation → Approximation algorithms analysis; Theory of computation → Randomized local search

Keywords and phrases
weighted connectivity augmentation, approximation, heuristic, integer linear program, algorithm engineering

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.11


Supplementary Material Software (Source Code): https://github.com/HeiConnect

Funding We acknowledge support by DFG grant SCHU 2567/3-1.

1 Introduction

Many real-world coherences can be modeled as graphs, including technological, social, and biological networks. A common problem of interest is the robustness of such a graph. Particularly in technological networks this is important for creating systems that are robust and fail-safe [12]. An example is a power grid where single lines can fail, either randomly due to age, or by targeted attacks. If a line fails, alternative routes are used which is increasing the load on them and therefore the chance of failure. To obtain a fail-safe network that can
survive both, random failures and targeted failures of important lines, the graph needs to be well-connected. Increasing the connectivity and therefore improving the robustness at minimum cost is known as connectivity augmentation or the survivable network problem. Another technological example is a computer network like the internet which should be designed in a fail-safe way while reliable transportation networks can avoid traffic congestion.

However, it is well-known that the weighted connectivity augmentation problem is NP-hard. Eswaran and Tarjan [8] have shown that the decision problem, whether there is an augmentation of at most a given weight, is NP-complete. Frederickson and Ja'Ja' [11] have shown that this is also true for the simpler special case where the graph is a tree, with weights being only 1 or 2. This justifies the importance of good heuristic and approximation algorithms. Furthermore, the weighted connectivity augmentation problem is APX-hard, which was also shown for the weighted tree augmentation problem by Kortsarz, Krauthgamer and Lee [21]. Despite the fact that there is no polynomial time approximation algorithm with an approximation factor arbitrarily close to 1, there has been much progress in improving the approximation ratio. Recently, the connectivity augmentation problem has been discussed frequently in the context of approximation algorithms with approximation factors below 2 [2, 27, 29, 30]. This includes work on special cases like the tree augmentation problem [29], as well as the general case [2, 30].

In recent years, Henzinger et al. [16, 17] developed the leading codes for the minimum cut problem in graphs. This includes the development of cutting-edge shared-memory inexact algorithms, consistently delivering near-optimal results. Additionally, they engineered state-of-the-art shared-memory exact algorithms [14], surpassing the previous state-of-the-art by an order of magnitude in running time, as well as highly efficient approaches for tackling the broader all minimum cut problem [18]. It turns out that these algorithms, i.e. computing minimum cuts, (enumeration of) all minimum cuts, and the efficient computation of a cactus representation of a graph, are important subroutines for algorithms that tackle the connectivity augmentation problem. Thus in this work, we heavily employ these recently developed techniques to engineer efficient algorithms for the connectivity augmentation problem.

Our Results. Our contribution in this work is two-fold. First, we give the first implementation and experimental evaluation of two recently discovered connectivity augmentation approximation algorithms due to Traub and Zenklusen [30]. More precisely, Traub and Zenklusen describe two algorithms: a greedy \((1 + \log 2 + \epsilon)\)-approximation and the local search based \((1.5 + \epsilon)\)-approximation, which we implement and evaluate.

Secondly, we propose three new heuristic and an exact algorithm. The first algorithm is a greedy heuristic considering link costs and the number of cuts covered by a link. This simple algorithm already outperforms all previous state-of-the-art algorithms by more than 22% improvement in solution quality (reduced link costs) on instances where links have small costs. Our second strategy uses minimum spanning trees to find a feasible solution first and then greedily improves it. Additionally, we present a local search algorithm that can improve a given solution by replacing link sets with cheaper ones. On instances with large link costs, the minimum spanning tree algorithm has the overall best performance regarding solution quality, running time and memory consumption. It computes solutions 8% better than the best performing previous state-of-the-art on these instances, while being a factor of 7 times faster on average. With our local search algorithm we can further improve these solutions on average by 2%. Lastly, we introduce a new exact solver using an ILP formulation with efficient cut enumeration as well as a fast initialization routine, for which we utilize our fast minimum spanning tree heuristic. Especially on real-world instances, it is able to outperform the previous state-of-the-art heuristic solvers regarding running time for both small and large link costs.
2 Preliminaries

An undirected graph $G = (V, E)$ is a structure that consists of a set of vertices $V$ and a set of edges $E \subseteq \binom{V}{2}$ connecting pairs of vertices. The number of vertices is denoted as $n$ and the number of edges as $m$. The graph $G$ is connected if there is a path between any two vertices. The edge connectivity of a graph is the maximal number of edge-disjoint paths that exist between any pair of vertices. A graph is $k$-edge-connected, if $k - 1$ arbitrary edges can be removed without disconnecting the graph. A partition of a graph is a partition of the vertex set into mutually disjoint sets.

A cut of a graph is a partition of the vertex set into two disjoint subsets, also called a bipartition. Any cut can be represented as one of its two constituent vertex sets. Every non-empty proper subset of $V$ is a cut. To prevent different representations of the same cut we use the notation where a cut is given as one set of the partition (only the representation that does not include an arbitrarily chosen root $r \in V(G)$ is used). The size or weight of a cut is the number of edges or the sum of the edge weights that have one endpoint in each subset. A cut is a minimum cut if there is no cut with smaller size or weight. The set of all minimum cuts is denoted as $C_G$ and cut : $C_G \times \binom{V}{2} \rightarrow \{0, 1\}$ is a function that is 1 if and only if the endpoints $u$ and $v$ of an edge $e = uv \in \binom{V}{2}$ lie in different sets of the partition of a cut $c \in C_G$.

The goal of the Weighted Connectivity Augmentation Problem (WCAP) is to increase the edge connectivity of a graph. More formally, for a given $k$-connected graph $G$ with a set of links $L \subseteq \binom{V}{2}$ and a cost function $c : L \rightarrow \mathbb{R}_{\geq 0}$, the task is to find the cheapest subset of links $S \subseteq L$ that will increase the edge connectivity to $k + 1$. A link $l \in L$ covers a minimum cut $c \in C_G$ if the size or weight of the cut $c$ is increased in the graph $G' = (V, E \cup \{l\})$. The graph $G_L = (V, L)$ is called the link graph. The set of links is disjoint with the set of edges, i.e. $L \cap E = \emptyset$. For the ease of notation the cost function is extended to sets, where it is the sum of the cost of all elements.

If the input graph is disconnected, the weighted connectivity augmentation problem coincides with the minimum spanning tree problem among its components. In this work we therefore only focus on connected graphs, as the other case is simple to solve via well-known efficient minimum spanning tree algorithms.

A cactus graph is a connected graph, such that any two cycles have at most one vertex in common. To distinguish between edges that lie within a cycle and those that do not, they are called cycle edges and tree edges, respectively. The cactus graph representation of the set of minimum cuts $C_G$ of a graph $G = (V, E)$ is a cactus graph $C = (V_c, E_c)$ with a function $\Pi : V \rightarrow V_c$ and its inverse $\Pi^{-1} : V_c \rightarrow 2^V$ which is defined as $v \mapsto \{u \in V : \Pi(u) = v\}$. The functions $\Pi$ and $\Pi^{-1}$ are defined such that each minimum cut in $C$ corresponds to a minimum cut in $G$, i.e. for all $c \in C_G : \bigcup_{v \in c} \Pi^{-1}(v) \in C_G$ and each minimum cut in $G$ is represented in the cactus graph, i.e. for all $c_G \in C_G$ there exists a cut $c_C \in C_C$ such that $\Pi(v) \in c_C$ for all $v \in c_G$. Figure 1 gives an example for a graph, with its minimum cuts and the corresponding cactus graph representation. Dinitz et al. [5] have shown that all minimum cuts of a connected graph $G = (V, E)$ can be represented as a cactus graph $C = (V_c, E_c)$. For more details on the computation of the cactus graph, we refer the reader to [18]. Analogous to the link graph $G_L$, we define the cactus link graph $C_L = (V_c, L_c)$. The set $L_c \subseteq L$ is defined as $L_c = \{(\Pi(v_1), \Pi(v_2)) : (v_1, v_2) \in L\}$ where we only keep the cheapest link between each vertex pair $(\Pi(v_1), \Pi(v_2))$. 


Related Work

This paper is a summary and extension of the master theses [24]. In this section, we present the state-of-the-art for computing all minimum cuts of a graph in the cactus graph representation, followed by research and implementations in the field of connectivity augmentation problems.

3.1 Minimum Cuts

Computing all minimum cuts is usually a fundamental step in connectivity augmentation. Nagamochi, Nakao and Ibaraki presented an efficient algorithm to compute all minimum cuts in the cactus graph representation [25]. A cactus representation can be computed in $O(mn + n^2 \log n + n^* m \log n)$ time where $n^*$ is the number of cycles in the cactus representation. They observed that all minimum cuts between two vertices $s$ and $t$ can be computed by running a maximum $s$-$t$-flow algorithm, and edges that are cut by no minimum cut can be contracted.

The current state-of-the-art algorithm to compute all minimum cuts is VieCut by Henzinger et al. [14, 18]. It uses linear time edge contraction based reduction rules and an optimized version of the algorithm by Nagamochi, Nakao and Ibaraki. For example, an edge $uv$ can be contracted if the connectivity between $u$ and $v$ is larger than the minimum cut. Such edges could be found by computing $k$ edge-disjoint spanning trees where $k$ is the size of the minimum cut [14, 26]. Furthermore, reduction rules by Padberg and Rinaldi [28] were adapted from the problem of finding one minimum cut to the problem of finding all minimum cuts. Lastly, edges that form a trivial minimum cut are contracted and remembered. These cuts are reintroduced at the end of the algorithm. The reduction rules are used exhaustively as long as a significant number of edges is contracted. The remaining kernel is solved based on the algorithm by Nagamochi, Nakao and Ibaraki [25]. In this work, we use VieCut [14, 18] to compute the cactus representation of a graph, which has a much better performance in practice than what the worst-case analysis predicts. Henceforth, when we analyse the complexity of an algorithm, we do not include the complexity to compute the cactus, as this is the same for all algorithms.

3.2 Connectivity Augmentation

There have been several approximation algorithms for the connectivity augmentation problem in the past. An early approach is using minimum cost arborescences, which was introduced by Frederickson and Ja’Ja’ [11] for bridge connectivity augmentation, the case where the graph is 1-connected but not 2-connected, and generalized for the WCAP by Watanabe et al. [32]. The bridge connectivity algorithm results in a 2-approximation while the generalization cannot guarantee an approximation factor. There are well-known 2-approximations for the WCAP. One possibility was discovered in 1992 and reduces the problem to a directed version by replacing each undirected edge with two directed edges [20]. The directed version can be
solved in polynomial time based on minimum-cost flows [10] or by using a linear program which has integral solutions for the cactus augmentation problem [3]. Another approach involves the LP relaxation of an ILP formulation combined with iterative rounding techniques [19].

Only recently, progress has been made on various approximation algorithms regarding special cases of the connectivity augmentation problem, as well as the general case. For the unweighted version of the connectivity augmentation problem the first approximation with factor below 2 was found in 2020 by Byrka et al. [2, 27]. The WCAP is reduced to the steiner tree problem, for which a specialized approximation gives an approximation factor of 1.91.

For the tree augmentation problem (TAP), where the cactus graph is a tree, and the unweighted connectivity augmentation problem an approximation factor of 1.393 was found in 2021 [3]. For the weighted tree augmentation problem, Traub and Zenklusen [29] discovered a \((1 + \ln 2 + \epsilon)\)-approximation, which builds upon the 2-approximation reducing the problem to a directed one and greedily improves this solution. Afterwards, they transferred the algorithm to the weighted connectivity augmentation problem and refined it to a \((1.5 + \epsilon)\)-approximation [30], which improves an arbitrary solution through local search. However, no implementations or experimental results of those algorithms existed until this point.

There has been recent work on randomized Monte Carlo algorithms that give a solution with high probability on graphs with integer edge weights based on maximum flow computations. The state-of-the-art is an \(\tilde{O}(m)\) time algorithm that gives a near-linear running time by Cen et al. [4]. This shows that the connectivity augmentation problem is simpler than the maximum flow problem as there is no known \(\tilde{O}(m)\) time maximum flow algorithm.

**Experimentally Evaluated Algorithms**

There have been practically applicable heuristic algorithms in the past, however, there has not been much progress on the WCAP recently. Watanabe et al. [23, 34, 33, 32] proposed five different approaches, called FSA, MW, FSM, SMC and HBD, including experimental evaluation. An observation used for all algorithms is that there is a subset of all vertices of the cactus graph representation that must be an endpoint in any augmentation. The algorithm FSA uses minimum cost arborescences based on the ideas of Frederickson and Ja’Ja’ [11]. MW is a 2-approximation also based on arborescences. FSM is based on maximum cost matchings. The third approach, SMC, is a greedy strategy adding the cheapest incident link for each vertex of the cactus graph representation. HBD is a combination of FSM and SMC. Experimental results showed that the solution quality of FSM is the best, followed by HBD, SMC, FSA and lastly MW [23, 32]. Regarding running time, SMC is the fastest algorithm, followed by FSA, HBD, FSM and lastly MW. HBD is considered the best general algorithm, because it prevents arbitrary bad solutions that may be produced by FSM or SMC. MW is the only algorithm with a guaranteed approximation factor, however, in practice it is slower and the solution quality of the other algorithms is better [23]. To the best of our knowledge, no other experimentally evaluated algorithms are mentioned in the literature.

**4 Approximation Algorithms**

We now describe the approximation algorithms implemented in this paper in more detail. First we describe the 2-approximation [3] followed by brief discussions on two approximation algorithms with approximation factors \((1 + \ln 2 + \epsilon)\) and \((1.5 + \epsilon)\) by Traub and Zenklusen [30] for which we present first implementations and experimental evaluations. For further, highly detailed information see [3, 30].
Engineering Weighted Connectivity Augmentation Algorithms

4.1 LP-based 2-Approximation

For the 2-approximation each undirected link $l = uv$ is replaced by two directed links $l_1 = (u, v)$ and $l_2 = (v, u)$, resulting in a set $L'$ with $|L'| = 2 \cdot |L|$. Then, the easier directed problem can be solved. Later, a directed solution is transferred to the undirected problem by replacing each link in the directed solution with the undirected one while removing duplicates. To solve the directed problem, a linear program based algorithm is implemented. This approach was first proposed by Jain [19], and Cecchetto, Traub and Zenklusen [3] provided a definition where the solution is integral for any cactus augmentation instance. Using Brand’s $O(N^2 \cdot 3.37 \log^2 N \log N/\delta)$ algorithm [31] where $N$ is the number of variables, the time complexity of the 2-approximation is $O(n^{4.74} \log^2 n \log n/\delta)$.

4.2 Relative Greedy $(1 + \ln 2 + \epsilon)$-Approximation

Traub and Zenklusen presented in [30] a greedy algorithm that improves upon the 2-approximation described above. It begins by exactly reducing a cactus graph representation to a ring graph and then replacing directed links in the solution with so called shadows, to form an arborescence. These shadows are links, that have the same weight as the original links and together still form an augmentation. The algorithm proceeds by greedily substituting sets of directed links with undirected ones resulting in mixed solutions. If all directed links are replaced by undirected ones, the solution is a solution to the original problem. The greedy objective is the ratio of the cost of the added undirected links and the cost of the directed links that are not needed anymore. Since this is difficult to compute, they only consider link sets that can be constructed iteratively by a dynamic program. Furthermore, it is easier to check if a given ratio is better or worse than the optimum. The algorithm uses binary search with respect to the ratio to determine the optimum along with a set of links that achieves this ratio. For each bisection a dynamic program is run.

The $(1 + \ln 2 + \epsilon)$-approximation algorithm has a polynomial running time, but it is computationally expensive, especially due to the complexity of the dynamic program involved. Let $\alpha = 4 \left\lfloor \frac{2}{\epsilon} \right\rfloor$, then, for graphs with $n > \alpha + 2$ the size of the dynamic programming table is $O(n^{2\alpha + 2})$. With this, the overall computational complexity of the algorithm for integer weighted problems is $O(n^{4\alpha + 7} \ln(OPT))$, where $OPT$ is the optimal augmentation weight. Note that the approximation ratio can only be improved if $1 + \ln 2 + \epsilon < 2$ requiring $\alpha \geq 28$.

4.3 Local Search $(1.5 + \epsilon)$-Approximation

The state-of-the-art approximation algorithm by Traub and Zenklusen [30] is a $(1.5 + \epsilon)$-approximation. A more detailed description and correctness proofs can be found in [30]. The algorithm is based on the ideas and the dynamic program of their relative greedy $(1 + \ln 2 + \epsilon)$-approximation described previously. The main difference is that the algorithm does not only greedily replace all links of a directed solution with undirected links. Instead, replaced links should iteratively improve the solution and can themselves be replaced in further iterations. The main part of this algorithm, the dynamic program, is the same as for the $(1 + \ln 2 + \epsilon)$-approximation. Therefore, the size of the dynamic programming table is bounded by $O(n^{2\alpha + 2})$. The total running time is bounded by $O(n^{4\alpha + 7}/\epsilon)$. In contrast to the $(1 + \ln 2 + \epsilon)$-approximation this is independent of the (upscaled) augmentation weight $OPT$.
5 Weighted Connectivity Augmentation Algorithms

This section describes our heuristic approaches and our exact algorithm for the WCAP. The data structures used for these algorithms are briefly described in Section 5.1. Unlike the approximation algorithms in the previous section, the heuristics cannot give guarantees on the solution quality, but aim at being fast or giving good solutions for many real-world cases. For all considered algorithms we only compute a solution on the cactus graph representation which can be transformed to the solution on the original graph, as stated in Theorem 1.

▶ Theorem 1 (Dinic, Karzanov and Lomonosov [5]). Let $G$ be a graph and $L$ the set of links. Furthermore, let $C$ be the corresponding cactus graph representation of $G$ and $C_L$ be the link graph of $C$. Then, a valid solution for the WCAP on $C$ is a valid solution to the WCAP on $G$.

5.1 Data Structures

All minimum cuts of a graph $G$ can be represented as a (potentially significantly smaller) cactus graph $C$, it is sufficient to do computations on the cactus graph $C$. To be able to give an augmentation for the original graph $G$, it is stored in adjacency list representation along with an array modeling the function $\Pi : V(G) \to V(C)$. Additionally, the link set $L$ must be transferred to a link set $L_C$ for the cactus graph $C$. A link $l = (u, v)$ is translated to a link $l_C = \Pi(l) := (\Pi(u), \Pi(v))$ in the cactus graph. To be able to reverse this function and obtain a link in $G$, the endpoints of the original link $l$ are stored as well. This can lead to parallel links $g, h \in L, g \neq h$ with $\Pi(g) = \Pi(h)$ in the link graph $G_L$ (with not necessarily equal weight). We drop all parallel links in $G_L$ except for one of smallest weight. We store the link set $L_C$ in an adjacency matrix and only keep one link of minimum weight per vertex pair.

Dynamic Cactus / Updating Cactus

For a dynamic cactus representation we use a similar approach for updating the cactus as proposed by Henzinger, Noe and Schulz in [15], where a union find data structure is used to keep track of the function $\Pi$ that associates each vertex of $G$ with a vertex of the cactus $C$.

When adding a link $l = (u, v)$ that crosses a minimum cut, the cactus shrinks. Dinitz has shown that cuts on the $uv$-path are affected [6]. In particular, vertices that lie on every $uv$-path in the cactus are contracted to update the cactus [15]. Those can be found by computing a path in the tree of cycles $T$. This is the graph that contains all cycles as vertices and has an edge between two vertices if the corresponding cycles share a vertex. Since $T$ is a tree, the paths are unique. For every cycle in the path, the shared endpoints are contracted. This graph $T$ is maintained by our data structure.

To be able to efficiently compute the number of minimum cuts that a given link crosses, this data structure is extended and modified. The naive approach of counting the number of cuts that a given link crosses evolves checking every possible cut and takes $O(|V_c|^2)$.

▶ Theorem 2. Using our data structure, we can compute the number of cuts that a given link crosses in $O(|V_c|)$.

Proof. A cut always cuts two edges of the same cycle, therefore each cycle can be considered separately. As a link $l = (u, v)$ affects all cuts on the $uv$-path [5], all cycles on the $uv$-path need to be considered. These cycles are found by a BFS in $T$ taking $O(|V_c|)$ time because $T$ is a tree and there cannot be more cycles than vertices in the cactus graph. Counting the
Algorithm 1 GWC.

\[ \text{Algorithm 1 GWC.} \]

**input** \( G = (V, E), L, c : L \to \mathbb{R}_{\geq 0} \)

**output** augmentation \( S \subseteq L \)

**procedure** GWC\((G, L, c)\)

\( C = (V_c, E_c) \leftarrow \text{cactusByVieCut}(G) \)

\( C_L = (V_c, L_c) \leftarrow \text{buildLinkGraph}(V_c, L) \)

\( S \leftarrow \emptyset \)

while \( L_c \neq \emptyset \) do

\( l \leftarrow \arg \min \{c(l)/\text{coveredCuts}(C, l) \mid l \in L_c\} \)

\( S \leftarrow S \cup \{l\} \)

\( C \leftarrow \text{updateCactus}(C, l) \)

\( L_c \leftarrow \{l \in L_c \mid l \text{ covers a mincut in } C\} \)

return \( S \)

number of cuts that the computed path crosses can be done in linear time too. For each edge in the path we go through all vertices contained in the corresponding cycle to compute the distance of the articulation points. As this is done at most once per cycle, we are able to compute the number of cuts that a given link crosses within \( O(|V_c|) \).

For further implementation details, we refer to [24].

5.2 Heuristic Algorithms

In this section we introduce a new heuristic \text{GreedyWeightCoverage} (GWC), where links are added to the solution greedily based on the costs per augmented cut, an algorithm based on minimum spanning trees called \text{MSTConnect} as well as a local search algorithm \text{LS}(k).

5.2.1 GWC

As the problem aims at minimizing the cost of an augmentation, it is natural to add links of small weight. At the same time, we want to minimize the number of links added. This leads to the heuristic GWC, where we greedily pick the link, for which the cost per augmented minimum cut, i.e. \( c(l)/a_l \) with \( a_l = \text{coveredCuts}(C, l) := \{|c \in C_G : \text{cut}(c, l) = 1|\} \) is minimal. If \( a_l = 0 \) the link \( l \) is not considered. This regards listing all minimum cuts in the cactus graph computed using VieCut \([14, 18]\).

A naive bound for the complexity of this is \( O(|V_c|^5) \), because the solution has at most \( O(|V_c|) \) links, and in each iteration the heuristic is computed for \( O(|L_c|) = O(|V_c|^2) \) links by checking \( O(|V_c|^2) \) minimum cuts. Using our custom data structure, see Section 5.1, we can run the algorithm in \( O(|V_c|^4) \).

The bound is very pessimistic and in practice the performance of the algorithm is much better, especially since we use efficient algorithms to enumerate all minimum cuts \([18]\).

We also tested other heuristics such as choosing the link with the smallest weight, such that at least one cut is covered, or picking an arbitrary uncovered minimum cut \( c \) and choosing the smallest weight link that covers \( c \). However, GWC produced significantly better results, which is why we only report details of this algorithm here.
Algorithm 2 MSTConnect.

\begin{algorithm}
\begin{algorithmic}
\State \textbf{input} \( G = (V, E) \), \( L \), \( c : L \rightarrow \mathbb{R}_{\geq 0} \)
\State \textbf{output} augmentation \( L' \subseteq L \)
\Procedure{MSTConnect}{(G, L, c)}
\State \( C = (V_c, E_c) \leftarrow \text{cactusByVieCut}(G) \)
\State \( C_L \leftarrow \text{buildLinkGraph}(V_c, L) \)
\State \( L_{MST} \leftarrow \text{MST}(C_L, c) \)
\State \( L' \leftarrow \text{sortDesc}(L_{MST}, c) \)
\For{\( l \in L' \)}
\State \If{\( \text{disposableLink}(l, C, L') \)}
\State \( L' \leftarrow L' \setminus \{l\} \)
\EndIf
\EndFor
\State \Return \( L' \)
\EndProcedure
\end{algorithmic}
\end{algorithm}

5.2.2 MSTConnect

After computing the cactus graph representation using VieCut [14, 18], the greedy strategies described above work by adding links to a set until this set is a valid connectivity augmentation. For MSTConnect, see Algorithm 2, we have a different approach. Here, we start with a (possibly much larger) set of links that increases the connectivity when added to the cactus graph. Then, we reduce this link set, while keeping a valid solution.

The complete set of links can have \( O(|V_c|^2) \) size. As checking if a link can be removed from a set can also be expensive, a small initial solution is essential. Using Theorem 3, an intuitive starting point is a minimum spanning tree (in case of incomplete link sets a minimum spanning forest) \( L_{MST} \) of the cactus link graph \( C_L \). For each link in this set, we check whether it can be removed in \( \text{disposableLinks} \) of Algorithm 2. Since we start with a MST the remaining links \( L' \) are always bounded such that \( |L'| < |V_c| \). Therefore, this process needs linear time, as shown in Theorem 4. If a link is disposable, we exclude it from the solution. Figure 2 (left) illustrates an example.

\begin{itemize}
\item \textbf{Theorem 3.} Let \( C = (V_c, E_c) \) be a cactus graph and \( C_L = (V_c, L_c) \) the link graph of \( C \), such that a feasible solution to the WCAP exists. Then, a minimum spanning forest \( L_{MST} \subset L_c \) for \( C_L \) is a feasible solution to the WCAP on \( C \).
\end{itemize}

\textbf{Proof.} See Appendix B.

\begin{itemize}
\item \textbf{Theorem 4.} Let \( G = (V, E) \) be a \( k \)-connected graph and \( C = (V_c, E_c) \) its cactus graph representation, \( L' \subset L \) an augmentation on the link graph \( C_L \) and \( l \in L' \). If \( |L'| < |V_c| \), then we can check if \( L' \setminus \{l\} \) is still an augmentation in \( O(|V_c|) \).
\end{itemize}

\textbf{Proof.} See Appendix B.

A minimum spanning forest on the cactus link graph \( C_L \) is computed using Kruskal’s algorithm with a complexity of \( O(m \log m) \) [22], where \( m \) is the number of links in \( C_L \). Checking if \( O(|V_c|) \) links can be removed from the solution is done in \( O(|V_c|^2) \), see Theorem 4. This results in an overall complexity of \( O(|V_c|^2 \log |V_c|) \) for MSTConnect.

As in GWC, we also tested the cost per augmented minimum cut \( c(l)/a_l \) as weight to compute the minimum spanning forest as well as using \( c(l)/a_l \) to sort the links to be removed in Algorithm 2. However, our experiments showed that using the weight \( c(l) \) of the links performed significantly better in both cases and thus the other versions are omitted.
Algorithm 3 LS(k).

**input** $G = (V, E)$, $L$, $c : L \to \mathbb{R}_{\geq 0}$, $S$

**output** improved augmentation $S' \subseteq L$

**procedure** $LS(k)(G, L, c, S)$

- $C = (V_c, E_c) \leftarrow \text{cactusByVieCut}(G)$
- $L' \leftarrow \text{reduceLinkSet}(C, L, c)$
- $X \leftarrow \text{getSwapCands}(L', S)$
- $S' \leftarrow S$

while $X \neq \emptyset$

- $(L_{in}, L_{out}) \leftarrow \text{bestCandidate}(X)$
- $X \leftarrow X \setminus (L_{in}, L_{out})$
- if $\text{isSwapValid}(L_{in}, L_{out})$

- $S \leftarrow \text{swap}(S', L_{in}, L_{out})$
- $X \leftarrow \text{getSwapCands}(L', S')$

return $S'$

5.2.3 Local Search

The core idea of our local search algorithm $LS(k)$ is the following. We want to remove links from our solution and replace them with a lighter set of links. The parameter $k$ limits the number of links within such a swap. Since these swaps can result in infeasible solutions, we have to check feasibility for each swap. This way a non-optimal solution $S \subset L$ can be improved. Next we describe the different steps of $LS(k)$ in detail, an overview is given in Algorithm 3.

We first compute the cactus graph representation $C$ of $G$ using VieCut [14, 18]. Afterwards, we reduce the link set $L$ to the union of $t$ disjoint minimum spanning forests in the link graph $C_L$. Here, the next minimum spanning forest is computed on $C_L$ after removing the links of the previously computed minimum spanning forest. This way the average degree is a small constant depending on $t$. We set $t = 2$, since larger values did not improve the performance.

The main idea of the remaining part in $LS(k)$ is the following. We search for non-solution links $L_{in} \subset L \setminus S$ and solution links $L_{out} \subset S$ with $|L_{in}| \leq k_{in}$, $|L_{out}| \leq k_{out}$ and $k_{in} + k_{out} \leq k$. If the swap $(L_{in}, L_{out})$ is feasible, we can create a new improved solution by swapping these sets, i.e. $S' := S \setminus L_{out} \cup L_{in}$. In $\text{getSwapCandidates}$ we compute possible swap candidates $(L_{in}, L_{out})$. These have to fulfill the following conditions:

- $L_{in} \cup L_{out}$ form an alternating path from $L_{in}$ and $L_{out}$ of length at most $k_{in} + k_{out} = k$
- $c(L_{in}) - c(L_{out}) < 0$, i.e. the swap improves the solution
- non-triviality, i.e. $\forall v \in V(L_{out}) : \{v\}$ is no minimum cut

We restrict the search for swaps to alternating paths. Since endpoints of these paths in $L_{out}$ lose an adjacent link, it is likely that a new adjacent link is necessary to keep the cut covered. Interior links of the path cover two vertices at once. Utilizing the specified criteria, we determine the sets $L_{in}$ and $L_{out}$ within the $\text{getSwapCandidates}$ function. This is achieved through an adapted depth-first search process, where traversal is restricted to alternate edges that are part of the solution and those that are not. The depth of the search is restricted by the parameter $k$. This modified search is executed starting at each vertex in $|V_c|$. At every vertex encountered, we verify the adherence to the second and third condition. If they are met, the path currently identified is included in the collection of potential swap candidates.
These conditions are not sufficient to form a feasible swap, they rather help to prune the set of possible candidates in advance. For the best candidate, i.e. the candidate where $c(L_{in}) - c(L_{out})$ is lowest, we check the validity of the swap in $\text{isSwapValid}$. Using a maximum flow approach, similar to MSTConnect, this check can be done efficiently as shown in Theorem 4. If the swap is valid, we swap links, recompute possible candidates and start over. If the swap is not valid, we check the next best candidate.

Figure 2 shows an example of how the LS(3) algorithm improves the solution computed by MSTConnect. We further improved the algorithm LS($k$) by integrating path caching, where we maintain vertex sequences of a path already checked using a hash table. Using Theorem 5, the running time for LS(3) is $O(|V_G|^3 \times \Delta^2 + |V_G|^4)$ and for LS(5) $O(|V_G|^4 \times \Delta^2 + |V_G|^5)$.

**Theorem 5.** LS($k$) has a complexity of $O(|V_G|^2(|V_G|^\frac{k}{2} \times \Delta^2 + |V_G|^\frac{k}{2}))$.

**Proof.** See Appendix B.

### 5.3 Exact Algorithm

We now describe an efficient implementation of an integer linear program for the problem, which can be used to solve (small) instances to optimality. The formulation is inspired by the linear program used for the 2-approximation in [3]. To the best of our knowledge, this formulation itself has not been used before. For the formulation of the WCAP, we introduce binary variables $x \in \mathbb{L}$ that decide which links are added to the augmentation. The objective given in (1) sums up the weight of all selected links. The constraint ensures that each minimum cut in the graph $G$ is increased by at least one link added to the augmentation.

\[
\min_x \sum_{l \in \mathbb{L}} x_l c(l) \quad \text{s.t.} \quad \sum_{l \in \mathbb{L}} \text{cut}(c, l)x_l \geq 1 \quad \forall c \in C_G; \quad x_l \in \{0, 1\}^{\mathbb{L}}.
\]  

(1)

Our ILP formulation is run on the cactus representation and uses efficient enumeration algorithms on it to list all minimum cuts $C_G$ of VieCut [18]. Moreover, our solver, which we call eILP, uses an initial solution computed by MSTConnect, which improves the running time by 5.5% on cycle graphs, which are the most difficult instances to solve with eILP. We choose MSTConnect since it is the fastest heuristic approach already giving good results (see Section 6).
6 Experimental Evaluation

We now experimentally evaluate the algorithms described in the previous sections. The approximation algorithms are evaluated and compared in terms of quality and running time in Section 6.2. Afterwards, our new proposed algorithms GWC, MSTConnect and MSTConnect+LS($k$) are compared to the state-of-the-art solvers HBD, FSM and SMC in terms of solution quality, running time and memory consumption.

6.1 Overview

6.1.1 Methodology

The experiments are run on a computer with an AMD EPYC 9754 128 core processor with 256 threads running at 3.1 GHz and 755 GB of main memory running Linux. The C++ code is compiled using gcc 11.4.0 with optimization level O3. The memory for each process is limited to 50 GB and the running time is limited to 3 hours. We run our algorithms on each instance with 5 different seeds to generate link costs as described below or in cases of generated cactus graphs we create 5 different graphs with the same number of vertices and cycles. The objective, i.e. the weight of the augmentation, the running time and the maximum memory used is measured. We use the geometric mean when averaging over different seeds or instances such that every instance has a comparable influence on the result.

Different algorithms are compared using performance profiles [7]. These plots use the best algorithm as baseline for each instance and relate the other algorithms to this baseline. A performance profile can use the objective function to compare quality, running time and memory consumption. The $x$-axis shows a parameter $\tau \geq 1$. On the $y$-axis the fraction of instances whose objective is at most $\tau \cdot \text{best}$ is plotted, in particular $\#\{\text{objective} \leq \tau \cdot \text{best}\} / \#\text{instances}$. For running time and memory usage, time and memory are used instead of the objective, respectively. At $\tau = 1$ the plot shows the fraction of instances where the algorithm is able to find the best solution / has the fastest running time or lowest memory consumption. Some algorithms are not able to solve every instance due to constraints on memory and time. We give more details in the respective sections. To solve the integer linear program in EILP and the linear program in the 2-approximation algorithm we used the Gurobi Optimizer [13]. The integer linear program solver uses pre-solving and the MSTConnect solution as initial solution.

6.1.2 Instances

The algorithms are evaluated using two different sets. The first set consists of three types of generated graph instances: cycles or ring graphs, stars and cactus graphs. Cycles and stars represent edge cases of cactus graphs, with an amount of minimum cuts between $O(|V_c|)$ and $O(|V_c|^2)$. To be able to test the algorithms on instances that represent more complex and larger cactus graphs, we generated cactus graphs by the algorithm described in Appendix A. All instances are listed in Table 1 including their properties. The second set are real-world instances. Many real-world graphs have unique or very few distinct minimum cuts, which leads to very small cactus graphs with only a few vertices. We used all connected graphs with non-trivial cactus graph representations from the 10th DIMACS Implementation Challenge [1] for which the cactus graph representation has at least 100 edges and at most 40000 vertices. In general, we use VieCut [18] to compute the cactus representation of a graph. We only compute the cactus representation once for every graph and henceforth only report running times of the algorithms when run on the cactus representation.
6.1.3 Link Sets

The algorithms can perform differently based on the distribution of the link costs. For instance, the performance of MSTCONNECT is affected due to different structured minimum spanning trees and lower cardinality of optimum solutions make matching-based approaches like FSM and HBD or our greedy heuristic GWC more efficient. Thus, we choose a uniform distribution in different intervals. First, we have a set of small link cost choosing the costs uniformly from the sets \{1, 2\}, \{1, \ldots, 9\} and \{1, \ldots, 99\} as used in [32, 34] to reproduce their results. The second set of large link costs consists of links chosen uniformly from the set \{1, \ldots, 100000\}. Since our algorithms need link costs within the interval [0, 1], we scale those costs by dividing through the largest occurring link cost. Furthermore, all instances have complete link sets, i.e. \(G = (V, E \cup L)\) is a complete graph.

6.2 Approximations

We first analyse the performance of the different approximation algorithms described in Section 4. Here, we can only consider very small graphs, as the approximation algorithms (see below) do not scale at all. For following experiments, we only report the results for \(\epsilon = 0.15\). This is due to the fact that \(\epsilon\) is only used to limit the number of considered links crossing cuts to \(4\left\lceil \frac{\alpha}{2} \right\rceil\) for the \((1.5 + \epsilon)\)-Approximation and \(4\left\lceil \frac{\alpha}{2} \right\rceil\) for the \((1 + \ln 2 + \epsilon)\)-Approximation. However, on small graphs the number of links that may cross a minimum cut is already limited by the number of vertices. Thus, although we tested values for \(\epsilon\) in [0.1, 0.5], it did not yield different result in terms of running time or link cost.

Figure 3 (left) shows a performance profile of the solution quality, for tiny graphs. As the optimal solution was computed by eILP, the \(x\)-axis gives the approximation ratio. The 2-approximation consistently gives the worst solutions, which means both, the \((1 + \ln 2 + \epsilon)\)-approximation and the \((1.5 + \epsilon)\)-approximation, can improve this initial solution.

Figure 3 (right) shows the running time of the approximation algorithms as well as our solver eILP with respect to the graph size. The graphs are cycle graphs with a complete set of links. The running time increases exponentially for the approximation algorithms. This is expected for graphs with fewer than \(\alpha \geq 28\) vertices as the computational complexity is exponential in \(4\min(n, \alpha) + 7\). Both, the \((1 + \ln 2 + \epsilon)\)-approximation and the \((1.5 + \epsilon)\)-approximation are orders of magnitude slower than the optimal integer linear program. Hence,
we do not consider them further when comparing against other state-of-the-art algorithms. The 2-approximation is slightly faster than eILP, but the difference is negligible and both can easily solve tiny graphs solvable by the dynamic program based approximations. Overall, we conclude that the approximations have very little relevance for solving connectivity augmentation problems and may only have theoretical value. Since the 2-approximation is able to solve larger graphs it is also used for the comparisons in the following.

6.3 State-of-the-Art Comparison

In this section we compare our algorithms GWC, MSTConnect and MSTConnect combined with LS(κ) as well as eILP against the performance of the 2-approximation algorithm as well as to the best, i.e. FSM and HBD, and the fastest, i.e. SMC, state-of-the-art solvers from Watanabe et al. [32, 33, 34]. For the algorithms from Watanabe et al. [32, 33, 34], neither the instances used in their experimental evaluation nor source code or binaries for the algorithms are available. Hence, we compare them against our implementation of these algorithms. For the comparison of solution quality, running time and memory consumption we give performance profiles for generated and real-world instances in Figure 4 (small link cost) and Figure 5 (large link cost).

General Remarks. First note that the 2-approximation algorithm yields the worst results (highest cost) on both data sets and link costs. Additionally, it is also the slowest and most memory consuming algorithm. The high memory consumption results from a) the reduction of undirected to directed links, which doubles the size of the link set which roughly gives factor of two and b) more importantly from running the linear program solver. We now analyse results for the remaining algorithms for different link costs and instances types separately.

6.3.1 Small Link Costs

The best performing heuristic in terms of solution quality on the graphs with small link costs is GWC, see Figure 4. Unlike the other algorithms, GWC uses the cost per augmented minimum cut to decide for the solution links. For small link costs, there are a lot of links having the same or similar costs. Therefore, the cost per augmented minimum cut is more impactful than relying only on the cost of a link. For small link costs the MSTConnect algorithms are outperformed by the competitors FSM and HBD regarding the solution quality. On the instances solvable by all algorithms, GWC achieves solutions that are on average only 4% larger than the optimum solution, while the results computed by the competitors FSM and HBD are on average 28% larger than the optimum solution value. The SMC algorithm computes worse results than any of our algorithms, performing only slightly better than the 2-approximation. Overall, our exact solver eILP can solve 85.6% of these instances to optimality. As can be seen in Figure 4 (middle), MSTConnect is the overall fastest algorithm. On generated instances with small link costs GWC, which yielded the overall best results on these graphs, is on average the second fastest algorithm for these instances. On real-world instances on the other hand, our exact solver eILP is on average more than 3 times faster than GWC and also outperforms all the previous state-of-the-art competitors regarding running time. Since the cactus graph representation of real-world instances usually do not have large cycles, this benefits our solver eILP. The number of minimum cuts and therefore the number of constraints of the ILP (1) grows quadratically with the size of the largest cycle and only linearly with tree edges.

1 We contacted the authors, however, we did not get an answer.
Figure 4 Performance profile for the state-of-the-art comparison on solution quality, running time and memory consumption on instances with small cost links.

6.3.2 Large Link Costs

The best (heuristic) results for graphs with large link costs are achieved by MSTConnect improved by LS($k$) for $k = 5$ closely followed by $k = 3$, see Figure 5 (top). On this set, the wide range of different link costs makes the cost per link more significant, which benefits MSTConnect. All competing algorithms HBD, FSM and SMC yield worse results compared to all our MSTConnect approaches including MSTConnect without local search and GWC. In particular, our fastest algorithm MSTConnect, is on average 8%
(restricted to the instances solvable by SMC) better compared to SMC, the best performing competitor on these instances, while also being 7 times faster. When comparing to the exact solution on the instances solvable by eILP, MSTConnect is on average 3.8% away from the optimal solution. When additionally using our local search to improve the solution, i.e. using LS(5), the result is only 1.8% away from the optimum solution. All of our heuristic algorithms outperform even the fastest competitor SMC with regard to running time on most large cost real-world instances. On all instances even our slowest algorithm GWC is 1.7 times faster than SMC, while our fastest algorithm MSTConnect outperforms SMC by
a factor of 6.3. When only considering real-world instances, the difference gets even larger. Here, MSTConnect is on average 9.1 times faster than SMC and even our exact solver eILP is a factor of 2.6 times faster than SMC.

6.3.3 Memory Consumption

MSTConnect requires overall the least amount of memory. Indeed, improving the solution with LS($k$) increases the memory consumption with increasing $k$, especially on small link cost instances. Still, the memory usage for MSTConnect + LS(5) is on average 1.8 times better than the memory consumption of the previous state-of-the-art algorithms. On average all our algorithms outperform the competitors regarding memory consumption on generated graphs. For the real-world instances only MSTConnect and MSTConnect + LS(5) (with large link costs also LS(5)), need less memory than the other algorithms on almost every instance. Overall, MSTConnect uses 2.6 times less memory than the competitor SMC with the lowest memory consumption. Lastly, we like to note that most of the real-world instances not solvable by eILP are only solvable with our MSTConnect approaches.

6.3.4 Advise for Practitioners

A critical factor for the algorithms is the cost associated with the links of the augmentation problem. When dealing with scenarios where the link costs are very large, it is advisable to lean towards MSTConnect algorithms. Conversely, for cases where the link costs are relatively small and instances are similar to the generated ones used in this work, the GWC algorithm emerges as a preferable choice. For other real-world instances with small link cost, the eILP method is the algorithm of choice. In general, local search can help to improve the result at the expense of running time.

7 Conclusion

In recent years, new scalable algorithms for the minimum cut and the all minimum cut problem have been engineered [16, 17]. These algorithms are important subroutines for algorithms that tackle the connectivity augmentation problem. This inspired us to engineer novel efficient algorithms for the connectivity augmentation problem. In this work, we implemented recently published approximation algorithms, as well as new heuristic strategies and an exact approach for solving the weighted connectivity augmentation problem in large graphs efficiently. Our greedy heuristic GWC excels in solving small link cost instances, while our minimum spanning tree-based algorithm MSTConnect is the top choice for large cost instances in terms of solution quality, running time, and memory consumption. Additionally, we introduce a novel local search algorithm LS($k$) that enhances existing solutions – the first local search algorithm in the literature. Lastly, we engineer an exact solver eILP, which is also able to compete regarding running time on real-world instances with small link costs. We conducted experiments comparing our implementations with our best faith implementation of Watanabe et al.’s state-of-the-art solvers. Our algorithms significantly surpass their results in solution quality, running time, and memory consumption and are very close to optimal solutions. Surprisingly, on real-world instances all previous state-of-the-art algorithms, i.e. the algorithm by Watanabe et al., are even outperformed in terms of running time by our exact solver eILP on the (large number of) instances that it could solve. Instances that the eILP approach could not solve, have also not been solved by the algorithms by Watanabe et al.. Important future work includes exploring additional local search algorithms and reducing its search space through additional pruning. Additionally, we want to improve the scalability of these algorithms even further.
References


Engineering Weighted Connectivity Augmentation Algorithms


A Cactus Graph Generation

This section describes an algorithm that is able to generate graphs with given properties, namely the number of vertices and the number of cycles.

A.1 Generating Cactus Graphs

Given two integers \( n, c \in \mathbb{N} \), \( n > c \), the goal is to generate a cactus graph \( C \) with \( n \) vertices and \( c \) cycles. This is done by generating the cycles iteratively. The average number of vertices per cycle is \( n/c \). To get a larger amount of possible graphs the number of vertices per cycle is randomly distributed around the average \( n/c \). A Poisson distribution turned out to yield a higher variety of graphs than a uniform distribution. To ensure that the correct number of cycles will be achieved, the distribution range is bounded such that, considering cycles already generated, at least one vertex for every remaining cycle is available. The first generated cycle is used as the base graph. Each consecutive cycle must additionally use an existing vertex to connect to the base graph. This vertex is chosen uniformly among existing vertices.

A.2 Graph with given Cactus Graph

Given a cactus graph \( C \), one might ask how a graph \( G \) of which \( C \) represents all minimum cuts could look like. Trivially \( G \) could be equal to \( C \). Different graphs could be constructed by reversing the process of edge contractions during the computation of a cactus graph. In particular, each vertex of the cactus graph \( C \) could be replaced by a dense subgraph. Let \( k \) be the desired connectivity. Then, each vertex can be replaced by an at least \( (k + 1) \)-connected subgraph while each link is replaced by \( k \) unweighted links in case of a tree edge or by \( k/2 \) links in case of a cycle edge between corresponding dense subgraphs. However, for all algorithms considered in this paper, neither the structure of the original graph nor the connectivity \( k \) matter as they are abstracted in preprocessing steps. Therefore, only the simplest case of cactus graphs with connectivity \( k = 2 \) is considered.

B Omitted Proofs

Theorem (3). Let \( C = (V_c, E_c) \) be a cactus graph and \( C_L = (V_c, L_c) \) the link graph of \( C \), such that a feasible solution to the weighted connectivity augmentation problem exists. Then, a minimum spanning forest \( L_{\text{MST}} \subset L_c \) for \( C_L \) is a feasible solution to the connectivity augmentation problem on \( C \).

Proof. If the graph \( C_L \) is connected, \( L_{\text{MST}} \) is a tree. In this case there is a path in \( C_L \) between any two vertices and therefore every cut in \( C \) is covered. Therefore, \( L_{\text{MST}} \) is a feasible solution to the weighted connectivity augmentation problem. Assume \( C_L \) is not connected and \( L_{\text{MST}} \) is not a feasible solution to the WCAP, i.e. the connectivity of \( \hat{C} = (V_c, E_c \cup L_{\text{MST}}) \) is not increased compared to the connectivity of \( C \). Then, there must be a cut \( c \) which is not covered by the links in \( L_{\text{MST}} \). However, since there exists a solution to the problem there must be a link \( l = (u, v) \) in \( C_L \) covering the cut \( c \). Since \( L_{\text{MST}} \) is a minimum spanning forest...
we can check if representation, complexity for most both endpoints can vary. This results in at most path, since the endpoints are fixed. If there are only $S$ current solution path of length exactly $k$ has to have at least $\lfloor \Delta^2 \rfloor 2k$ links, since these are the dominating factor. Each path of length exactly $k$ has to have at least $k_1 = \lfloor \Delta^2 \rfloor 2k$ and at most $k_2 = \lceil \Delta^2 \rceil 2k$ edges from the current solution $S$. There can be $\mathcal{O}(|V_c|^2k_1 + |V_c|^2k_2)$ swaps in $X$ using different links from $S$. Now we estimate the number of different paths using the same links from $S$. Let $k$ be odd and there are $k_3$ equal solution links in the path. Then, there can only be one such path, since the endpoints are fixed. If there are only $k_1$ equal solution links in the path, then both endpoints can vary. This results in at most $\Delta^2$ different paths, where $\Delta$ is the max degree in $(V_c, L' \cup S)$. When $k$ is even, one endpoint of the path is free and there are at most $\Delta$ different paths. Overall, this yields $|X| \in \mathcal{O}(|V_c|^2 \lfloor \Delta^2 \rfloor \times \Delta^2 + |V_c|^2\lfloor \Delta^2 \rfloor)$ and results in a complexity for $LS(k)$ of $\mathcal{O}(|V_c|^2(|V_c|^{\frac{3}{2}} \times \Delta^2 + |V_c|^{\frac{3}{2}}))$. ▶

**Theorem (5).** $LS(k)$ has a complexity of $\mathcal{O}(|V_c|^2(|V_c|^{\frac{3}{2}} \times \Delta^2 + |V_c|^{\frac{3}{2}})).$

**Proof.** The running time for $LS(k)$ is dominated by the while loop in Algorithm 3. We have at most $|V_c|$ re-computations of the set $X$. Therefore, we need to check validity for at most $|V_c| \times |X|$ swaps. This results in a running time of $\mathcal{O}(|V_c|^2 \times |X|)$. To estimate $|X|$ we only consider the swaps using exactly $k$ links, since these are the dominating factor. Each path of length exactly $k$ has to have at least $k_1 = \lfloor \Delta^2 \rfloor$ and at most $k_2 = \lceil \Delta^2 \rceil$ edges from the current solution $S$. There can be $\mathcal{O}(|V_c|^2k_1 + |V_c|^2k_2)$ swaps in $X$ using different links from $S$. Now we estimate the number of different paths using the same links from $S$. Let $k$ be odd and there are $k_3$ equal solution links in the path. Then, there can only be one such path, since the endpoints are fixed. If there are only $k_1$ equal solution links in the path, then both endpoints can vary. This results in at most $\Delta^2$ different paths, where $\Delta$ is the max degree in $(V_c, L' \cup S)$. When $k$ is even, one endpoint of the path is free and there are at most $\Delta$ different paths. Overall, this yields $|X| \in \mathcal{O}(|V_c|^2 \lfloor \Delta^2 \rfloor \times \Delta^2 + |V_c|^{\frac{3}{2}})$ and results in a complexity for $LS(k)$ of $\mathcal{O}(|V_c|^2(|V_c|^{\frac{3}{2}} \times \Delta^2 + |V_c|^{\frac{3}{2}})).$ ▶
### C Instance Details

**Table 1** Properties of all graphs $G$ and their corresponding cactus graph $C$ used in the evaluation.

| class         | graph                          | $|V(C)|$ | $|E(C)|$ | $|V(G)|$ | $|E(G)|$ |
|---------------|-------------------------------|--------|--------|--------|--------|
| Real-World    | coAuthorsCiteseer             | 30,322 | 30,321 | 227,320 | 814,134 |
|               | preferentialAttachment        | 28,530 | 28,529 | 100,000 | 499,985 |
|               | delaunay_n21                  | 23,719 | 23,718 | 2,097,152 | 6,291,408 |
|               | luxembourg.osm               | 23,077 | 23,076 | 114,599 | 119,666 |
|               | kkt_power                     | 22,388 | 22,387 | 2,063,484 | 6,482,320 |
|               | delaunay_n20                  | 11,740 | 11,739 | 1,848,576 | 3,145,686 |
|               | coPapersDILP                 | 10,244 | 10,243 | 540,486 | 15,245,729 |
|               | as-22july06                   | 7,999  | 7,998  | 22,963 | 48,436  |
|               | hugenetic-00000               | 6,617  | 8,049  | 5,824,554 | 8,733,523 |
|               | coPapersCiteseer              | 6,372  | 6,371  | 434,102 | 16,686,720 |
|               | delaunay_n19                 | 5,977  | 5,976  | 524,288 | 1,572,833 |
|               | PGPocietytomo                 | 5,513  | 5,512  | 10,680 | 24,316  |
|               | vsp_vibrobox_sca7r7-2c_rilldd | 3,956  | 3,955  | 77,328 | 435,586  |
|               | vsp_finan512_sca7r7-2c_rilldd | 3,936  | 3,935  | 139,752 | 552,920  |
|               | vsp_setapl-2b_and_seymourl    | 3,288  | 3,287  | 40,174 | 140,831  |
|               | finas012                      | 3,073  | 3,072  | 74,752 | 261,120  |
|               | delaunay_n18                 | 2,929  | 2,928  | 262,144 | 786,396  |
|               | vsp_south3l1_splatk           | 2,710  | 2,709  | 39,668 | 189,914  |
|               | vsp_model11_crew1_cr42_south3l| 2,561  | 2,560  | 45,101 | 189,976  |
|               | vsp_c-30_data_data           | 1,768  | 1,767  | 11,023 | 62,184   |
|               | power                        | 1,612  | 1,611  | 4,941 | 6,594    |
|               | delaunay_n17                 | 1,484  | 1,483  | 131,072 | 393,176  |
|               | af_shell9                    | 1,276  | 1,275  | 504,855 | 8,542,910 |
|               | vsp_bump2_c18_a011_model1_crew1| 1,212 | 1,211 | 56,438 | 300,801 |
|               | t68k                         | 1,136  | 1,135  | 60,005 | 89,440   |
|               | vsp_p02911_seymourl_iiasa     | 942    | 941    | 10,498 | 53,868   |
|               | ldooor                       | 904    | 903    | 952,203 | 227,851,136 |
|               | latin_square_10              | 901    | 900    | 900    | 307,350  |
|               | delaunay_n16                 | 790    | 789    | 65,536 | 196,575  |
|               | add32                        | 681    | 680    | 4,900  | 9,462    |
|               | vibrobox                     | 625    | 624    | 12,328 | 165,250  |
|               | add20                        | 367    | 366    | 2,395  | 7,462    |
|               | delaunay_n15                 | 358    | 358    | 32,768 | 98,274   |
|               | NLR                          | 336    | 335    | 4,163,763 | 12,487,976 |
|               | G3_circuit                   | 317    | 316    | 1,585,478 | 3,037,674 |
|               | vsp_befref_fxm_2_4_air02     | 215    | 214    | 14,109 | 98,224   |
|               | delaunay_n14                 | 181    | 180    | 16,384 | 49,122   |
|               | auilikw1                     | 163    | 162    | 943,695 | 38,354,076 |
|               | email                        | 156    | 155    | 1,133  | 5,451    |
|               | uk                           | 136    | 135    | 4,824  | 6,847    |
|               | M6                           | 132    | 131    | 3,501,776 | 10,501,936 |
|               | cage15                       | 129    | 128    | 5,154,859 | 47,022,346 |
|               | memplus                      | 110    | 109    | 17,758 | 54,196   |

**Generated Cycles**

| cycle-5000 | 5,000 | 5,000 |
| cycle-1000 | 1,000 | 1,000 |
| cycle-500  | 500   | 500   |
| cycle-300  | 300   | 300   |
| cycle-200  | 200   | 200   |
| cycle-100  | 100   | 100   |
| cycle-50   | 50    | 50    |

**Generated Stars**

| star-5000 | 5,000 | 4,999 |
| star-1000 | 1,000 | 999   |
| star-500  | 500   | 499   |
| star-300  | 300   | 300   |
| star-200  | 200   | 199   |
| star-100  | 100   | 99    |
| star-50   | 50    | 49    |

**Generated Cacti**

| cactus[16-20] | 1000 | 1199 |
| cactus[06-10] | 100  | 119  |
| cactus[01-05] | 100  | 109  |
Graph algorithms with polynomial space and time requirements often become infeasible for massive graphs with billions of edges or more. State-of-the-art approaches therefore employ approximate serial, parallel, and distributed algorithms to tackle these challenges. However, such approaches require storing the entire graph in memory and thus need access to costly computing resources such as clusters and supercomputers. In this paper, we present practical streaming approaches for solving massive graph problems using limited memory for two prototypical graph problems: maximum weighted matching and minimum weighted edge cover. For matching, we conduct a thorough computational study on two of the semi-streaming algorithms including a recent breakthrough result that achieves a $1/(2 + \varepsilon)$-approximation of the weight while using $O(n \log W/\varepsilon)$ memory (here $n$ is the number of vertices and $W$ is the maximum edge weight), designed by Paz and Schwartzman [SODA, 2017]. Empirically, we show that the semi-streaming algorithms produce matchings whose weight is close to the best $1/2$-approximate offline algorithm while requiring less time and an order-of-magnitude less memory.

For minimum weighted edge cover, we develop three novel semi-streaming algorithms. Two of these algorithms require a single pass through the input graph, require $O(n \log n)$ memory, and provide a 2-approximation guarantee on the objective. We also leverage a relationship between approximate maximum weighted matching and approximate minimum weighted edge cover to develop a two-pass $3/2 + \varepsilon$-approximate algorithm with the memory requirement of Paz and Schwartzman’s semi-streaming matching algorithm. These streaming approaches are compared against the state-of-the-art 3/2-approximate offline algorithm.

The semi-streaming matching and the novel edge cover algorithms proposed in this paper can process graphs with several billions of edges in under 30 minutes using 6 GB of memory, which is at least an order of magnitude improvement from the offline (non-streaming) algorithms. For the largest graph, the best alternative offline parallel approximation algorithm (GPA+ROMA) could not finish in three hours even while employing hundreds of processors and 1 TB of memory. We also demonstrate an application of semi-streaming algorithm by computing a matching using linearly bounded memory on intersection graphs derived from three machine learning datasets, while the existing offline algorithms could not complete on one of these datasets since its memory requirement exceeded 1TB.
1 Introduction

Solving large-scale graph problems is at the forefront of various research fields, such as high performance computing, data science, and machine learning. An attractive computational model for large-scale graph computations is the semi-streaming model \cite{14, 35}, which promises to require significantly lower space than the traditional offline algorithms. In this paper, we study maximum weight Matching (\textit{MWM}) and minimum weight edge cover (\textit{MWC}) in the semi-streaming model. A matching (edge cover) in a graph is a subgraph where the degree of each vertex has an upper (lower) bound of one. On weighted graphs, for the \textit{MWM}, we seek a subgraph with the maximum sum of weights of the edges, while for the \textit{MWC} we minimize the sum of weights.

Matchings have been heavily researched in combinatorial optimization as they have rich algorithmic structures with many real-world applications. \textit{MWM} can be solved in polynomial time, but the optimal algorithms are expensive and not suited for graphs with billions of edges or more. Consequently, for the past twenty years, offline approximation algorithms have been developed for the \textit{MWM} problem \cite{3, 40, 8, 37, 9}. See \cite{32, 39} for surveys and a computational study of these algorithms. In the semi-streaming model, matching has also been extensively studied. The first semi-streaming algorithm for \textit{MWM} is $1/6$-approximate, and is due to Feigenbaum et. al. \cite{14} (FB, henceforth). After several improvements, an algorithm with the approximation ratio of $1/2+\varepsilon$ was designed by Paz and Schwartzman \cite{36} (PS, henceforth). Both algorithms use only $O(n \text{ polylog}(n))$ bits of space, where $n$ is the number of vertices of the graph.

Several natural questions arise: are these streaming algorithms memory-efficient in practice? By optimizing memory, do they sacrifice quality or runtime compared to offline approximation algorithms? In this paper we conduct a thorough computational study of two of the streaming matching algorithms (FB and PS) and compare them with four representative offline approximation algorithms (Greedy \cite{3}, PGA \cite{8}, GPA \cite{32} and ROMA \cite{37}). Our experiments reveal that the streaming algorithms (especially PS) stand out in terms of quality, memory, and runtime. We also show that the post-processing phase of the PS algorithm can be made parallel using a locally dominant strategy.

Edge cover has not been studied in the semi-streaming model, while in offline settings, there exist several approximation algorithms \cite{27, 17, 16, 39}. For edge cover, we develop and implement three new semi-streaming algorithms (\textit{NN}, \textit{OnePass} and \textit{TwoPass}) with approximation guarantees of $3/2+\varepsilon$ and $2$. These algorithms are compared against a state-of-the-art offline algorithm, the primal-dual algorithm (PD \cite{17}), which is $3/2$-approximate.

We also consider an application for \textit{MWM}, \textit{MWC} and variants: construct sparse graph representations of large datasets, where each instance has several features. Such data sets are used for downstream applications such as semi-supervised learning \cite{25} and privacy preservation \cite{26}. A common practice here is to construct a complete graph with instances as the vertices and edges weighted by a similarity or dissimilarity measure computed from the features of their endpoints. This graph is then sparsified by selecting a subset of edges using variants of matching or edge cover. But this limits the size of data one can process since the complete graph has $O(n^2)$ edges for $n$ instances. This motivates the use of streaming algorithms, where we do not need to store the graph as it is generated but only a small footprint of it. Indeed, for one of our test problems, the offline algorithm runs out of memory, whereas for two others, we reduce the memory used by a factor of more than 100.
2 Preliminaries

Notations. Let $G = (V, E, w)$ be a simple undirected graph with vertex set $V$ and edge set $E$, and let $w : E \rightarrow \mathbb{R}_{\geq 0}$ be a positive weight function defined on the edges. We denote $|V| = n$ and $|E| = m$ throughout the paper. Let $W_{\text{max}}$ denote $\max_{e \in E} \{w(e)\}$, and $W_{\text{min}}$ denote $\min_{e \in E} \{w(e)\}$. We assume both $W_{\text{max}}$ and $W_{\text{max}}/W_{\text{min}}$ are $\mathcal{O}(n^\alpha)$, where $\alpha$ is a positive real-valued constant. This assumption lets us represent the weights and their sum in $\mathcal{O}(\log n)$ bits. An edge subset $F \subseteq E$ induces a subgraph of $G$ with the endpoints of $F$ as vertices and edges from $F$. We use $N_F(\cdot)$ to denote the neighborhood of a vertex or edge in the subgraph induced by $F$. For a vertex $v$, $N_F(v)$ may represent the edges incident on $v$ ($\{e \in E : v \cap e \neq \emptyset\}$) or vertices adjacent to $v$ ($\{u \in V : \{u, v\} \in E\}$), and which definition is used will be clear from the context. Similarly, we define $N_F(e)$ for an edge $e$. If $f$ is a function and $X$ is a set of elements on which $f$ is defined, then $f(X) = \sum_{e \in X} f(e)$.

\textbf{MWM and MWC Problem.} Given a graph $G = (V, E, w)$, a matching is a subset of edges $M \subseteq E$, where every vertex of $G$ has at most one endpoint in $M$. A maximum weight matching (MWM) is a matching $M^*$ of maximum sum of weights $w(M^*)$ among all matchings. In an edge cover $C \subseteq E$, every vertex of $G$ has at least one endpoint in $C$. A minimum weight edge cover (MWC) is an edge cover $C^*$ of minimum sum of weights $w(C^*)$. The primal-dual formulation of the MWM problem is shown in Equations (1) and (2).

\begin{align*}
\max & \quad \sum_{e \in E} w(e)x(e) \\
\text{s.t.} & \quad \sum_{e \in N(v)} x(e) \leq 1 \forall v \in V \\
& \quad x(e) \geq 0 \forall e \in E. \tag{1} \\
\min & \quad \sum_{e \in E} y(e) \\
\text{s.t.} & \quad y(u) + y(v) \geq w(e) \forall e = (u, v) \in E \tag{2} \\
& \quad y(v) \geq 0 \forall v \in V.
\end{align*}

The Semi-Streaming Model. For the semi-streaming MWM and MWC problems, in a pass, the edges of $E$ are presented one at a time in an arbitrary order. We aim to compute a matching or edge cover in $G$ at the end of the stream, using small memory and few passes. The semi-streaming algorithm is output-sensitive, i.e., it is allowed to use memory sizes for processing that are proportional (up to a polylog factor) to the size of the memory needed to store the output. For MWM and MWC, the final solution size is $\mathcal{O}(n)$, and hence the memory limit is $\mathcal{O}(n \text{ polylog}(n))$.

3 Related Work

Henzinger et al. first considered streaming graph problems, using sublinear space in [23]. Unfortunately, many graph problems, such as connectivity and finding paths, are provably intractable with sublinear space. Hence the Semi-streaming model [14, 35] was introduced, which relaxes the memory limit to $\mathcal{O}(n \text{ polylog}(n))$, where $n$ is the number of vertices of the graph. Most of the semi-streaming graph algorithms are analyzed in either an arbitrary or random order of arrival of the elements. A random order stream assumes the streaming elements arrive uniformly at random, whereas in arbitrary order, the arrival of the stream could be adversarial. An early survey of semi-streaming graph problems is included in [34].
Algorithm 1 PS Matching(e\{u, v\}, \varepsilon).

Input: Stream of edges e\{u, v\}, A constant \varepsilon
Output: Matching M, using \(O(n \log^2 \frac{1}{\varepsilon})\) bits space

1: \(\forall v \in V : \phi(v) = 0; S \leftarrow \emptyset; M \leftarrow \emptyset\)  \(\triangleright \) Initialization
2: \(\triangleright \) Stream Process
3: for \(e(u, v) \in E\) do
4: if \(w(e) > (1 + \varepsilon)(\phi(u) + \phi(v))\) then
5: \(w'(e) = w(e) - (\phi(u) + \phi(v))\)
6: \(\phi(u) = \phi(u) + w'(e); \phi(v) = \phi(v) + w'(e)\)
7: \(S . \text{push}(e)\)
8: \(\triangleright \) Post Processing
9: while S is not empty do
10: \(e(u, v) \leftarrow S . \text{pop}()\)
11: if \((V(M) \cap \{u, v\}) = \emptyset\) then
12: \(M \leftarrow M \cup \{e\}\)

Matching problems are an active area of research in the semi-streaming model. For arbitrary edge arrival, Feigenbaum et al. [14] first studied semi-streaming $\text{MWM}$ and presented a $1/6$-approx matching, which was improved by several authors over time: [33, 42, 12, 6]. In a recent breakthrough, Paz and Schwartzman [36] showed that a simple algorithm could achieve a $1/(2 + \varepsilon)$-approximation. Ghaffari and Wajc further simplified their algorithm and analysis in [19]. For arbitrary order of stream, this result remains the best till now. For random arrival, Gamlath et al. [18] broke the $1/2$-approximate barrier and showed a $(1/2 + \varepsilon)$-approximate algorithm in a single pass.

For the $\text{MWC}$ problem, we are unaware of any semi-streaming algorithms in the literature. An edge cover can be formulated as a special case of a set cover problem, and for weighted set cover, Emek and Rosen [11] designed a single pass semi-streaming algorithm with approximation ratio $\sqrt{n}$. However, we report constant factor approximation algorithms for edge cover.

Software for modeling streaming graphs includes STINGER [10], graph-stream [29], and GraphStream [38]. Recent algorithmic and computational studies on streaming combinatorial problems include set-cover [4], connected components [41], and hypergraph partitioning [13]. However, we are not aware of any implementation and experimental study of streaming matching algorithms in the literature. Recent work on engineering algorithms for the dynamic matching problem with computational evaluations is presented in [1, 2, 21, 22].

4 Semi-streaming $\text{MWM}$ Algorithms

We briefly describe the two semi-streaming $\text{MWM}$ algorithms we implemented. Feigenbaum et al. [14] developed the first semi-streaming $\text{MWM}$ algorithm (FB), which maintains a matching $M$, and modifies it when a sufficiently heavy edge arrives. When an edge $e\{u, v\}$ arrives, FB inspects whether \(w(e) > 2 \cdot w(N_M(e))\), where $N_M(e)$ represents the neighboring edges of $e$ already in $M$. If the inequality is true, $e$ is inserted into $M$, and the edges in $N_M(e)$ are removed; otherwise, $e$ is ignored. The breakthrough $\frac{1}{2+\varepsilon}$-approximate semi-streaming algorithm (PS) for maximum weighted matching is due to Paz and Schwartzman [36]. The original algorithm was analyzed using local ratio techniques, but later Ghaffari and Wajc [19] provided a simpler primal-dual based analysis of the algorithm which we adopt here.

We now describe the idea underlying the Paz and Schwarzman algorithm. Consider the non-streaming setting first. The algorithm chooses an edge with positive weight, includes it in a stack for candidate matching edges, and subtracts its weight from neighboring edges. It
repeats this process as long as edges with positive weights remain. At the end, we unwind the stack and greedily add edges in the stack to the matching. This means that once an edge is added to the matching, any neighboring edges in the stack cannot be added to the matching. To adapt this algorithm to the streaming setting, we keep an approximate dual variable for each vertex that accumulates the weight of edges incident on the vertex and added to the stack. When an edge arrives, we subtract the sum of dual variables of the endpoints of the edge from its weight, and if this reduced weight is positive, then it is added to the stack. If not, it is discarded. The rest of the algorithm proceeds as before. To bound the size of the stack to $O(n \log n)$, we need one more idea, which is to include an edge in the stack only if its weight is greater than a constant factor $(1 + \varepsilon)$ times the sum of the approximate dual variables. It can be shown that if the edge weights are polynomial in $n$, then the size of the stack is bounded as desired, and that the approximation ratio becomes $1/(2 + \varepsilon)$.

A formal description of the method is shown in Algorithm 1. It initializes the approximate dual variables (the vector $\phi$) to zero, and then processes the streaming edges one by one. When an edge $e$ arrives, the algorithm decides whether to store it in the set of candidate matching edges (the Stack $S$) or to discard it. This decision is based on whether the dual constraint (shown in the Algorithm) is approximately satisfied for this edge. If the edge is stored, we compute the reduced weight $w'(e) = w(e) - (\phi(u) + \phi(v))$ and add it to both $\phi(u)$ and $\phi(v)$. Ghaffari and Wajc [19] showed that as edges incident on a vertex $v$ are inserted into the stack $S$, they have weights that exponentially increase with the factor $1 + \varepsilon$. Thus for each vertex at most $O(\log_{1+\varepsilon} W_{\max}) = O(\log \frac{W_{\max}}{\varepsilon}) = O(\frac{\log n}{\varepsilon})$ edges are stored in $S$ (since we assume weights to be polynomial in $n$).

In the post-processing phase, the algorithm unwinds the stack and constructs a maximal matching, processing edges in the stack order, in serial. The post-processing phase of PS can be parallelized by computing a locally dominant matching that preserves the approximation guarantee, which we discuss next.

### 4.1 Post-processing using Locally Dominant Algorithm

We now show that the post-processing phase of the PS algorithm (Algorithm 1) can be made parallel using locally dominant matchings. To do so, we maintain an array of $n$ stacks $S$ instead of a single stack $S$. In the streaming phase, for the eligible incident edges of a vertex $v$, $S[v]$ stores the neighboring vertices of $v$. When an edge $\{u, v\}$ arrives if it is not discarded, we push $v$ in $S[u]$ and $u$ in $S[v]$. Let $E_S$ be the set of edges stored in $S$, i.e., $E_S := \bigcup_{u \in V} \bigcup_{v \in S[u]} \{u, v\}$.

In the post-processing phase, we use $S$ to compute a maximal matching locally. Given a set of matching edges $M$, let $V(M)$ be the union of the endpoints of $M$, i.e., $V(M) = \bigcup_{\{u, v\} \in M} \{u, v\}$. An edge $\{u, v\}$ is available w.r.t a matching $M$ if $e$ can be added to $M$ without violating any matching constraints, i.e., $V(M) \cap \{e, v\} = \emptyset$. Given a matching $M$, we say an edge $\{u, v\} \in E_S$ is locally dominant if the edge is available with respect to $M$ and $u$ is on the top of $S[v]$ and $v$ is on the top of $S[u]$. The matching $M$ is locally dominant if every edge of $M$ is locally dominant when added to the matching. When the streaming phase ends, we compute a locally dominant matching in the subgraph induced by the edges in $E_S$.

This new post-processing does not change the approximation guarantee of the original algorithm. We reuse an observation and a corollary from [19].

**Observation 1.** For $v \in V$, $y(v) = (1 + \varepsilon)\phi(v)$ is feasible in the dual LP (2).
Algorithm 2 Semi-Stream Matching(\{e\{u,v\}\}, \varepsilon).

**Input:** Stream of edges \(e\{u,v\}\), a constant \(\varepsilon\)

**Output:** \(\frac{1}{2(1+\varepsilon)}\)-approximate Matching \(M\), using \(O(n \log^2 n/\varepsilon)\) bits space

1: \(\forall v \in V: \phi(v) = 0\); \(M \leftarrow \emptyset\) \hspace{1cm} \text{\textbf{▷ Initialization}}
2: \(S \leftarrow\) Array of \(n\) empty stacks
3: for \(e\{u,v\} \in E\) do \hspace{1cm} \text{\textbf{▷ Stream Process}}
   4: if \(w(e) > (1 + \varepsilon)(\phi(u) + \phi(v))\) then
   5: \(w'(e) = w(e) - (\phi(u) + \phi(v))\)
   6: \(\phi(u) = \phi(u) + w'(e);\ \phi(v) = \phi(v) + w'(e)\)
   7: \([S[u]\).push(v); \([S[v]\).push(u)\]

8: while \(\exists e\{u,v\} \in E_S: e\) is locally dominant do \hspace{1cm} \text{\textbf{▷ Post Processing}}
9: \(M \leftarrow M \cup \{e\}\)
10: Remove \(u\) from \([S[v]\) and \(v\) from \([S[u]\)

**Corollary 2.** Let \(OPT(LP)\) denote the weight of an optimal solution of the LP (1). Any optimal matching \(M^*\) satisfies \(w(M^*) \leq OPT(LP) \leq (1 + \varepsilon)\sum_{v \in V} \phi(v)\).

Let \(\Delta\phi(e)\) for an edge \(e\) be the change to \(\sum_v \phi(v)\) in Lines 6 in the Algorithm 2. We observe that if we decide to push the endpoints of \(e\) into \(S\), then the change in the total approximate dual (\(\sum_{v \in V} \phi(v)\)) due to \(e\) is \(\Delta\phi(e) = 2w'(e)\). Otherwise, \(\Delta\phi(e) = 0\). At the end of the streaming phase, \(\sum_{e \in E} \Delta\phi(e) = \sum_{e \in V} \phi(v)\).

The following Lemma proves a relationship between the weight of an edge \(e\) and the reduced weight of all of its neighbors in \(E_S\) that occur in the stream before \(e\).

**Lemma 3.** Let the preceding edges of an edge \(e\{u,v\} \in E_S\) be denoted by \(P(e) = \{e'|e' \cap e \neq \emptyset, e'\) is inspected no later than \(e\}\). Then \(w(e) \geq \frac{1}{2} \sum_{e' \in P(e)} \Delta\phi(e') = \sum_{e' \in P(e)} w'(e')\).

**Proof.** Before inspecting \(e\) in the stream, let the values of \(\phi(u)\) and \(\phi(v)\) be \(\phi'(u)\) and \(\phi'(v)\), respectively. Let us first consider \(\phi'(u)\), which is the accumulation of all the updates \(w'(\{u,x\}) = \frac{1}{2}\Delta\phi(\{u,x\})\) for some vertex \(x\), where the edge \(\{u,x\}\) arrives before \(e\) in the stream. Hence, \(\{u,x\}\) is in \(P(e)\), and \(v\) is positioned higher than \(x\) in \([S[v]\). A similar argument also holds for the edges \(\{v,x\}\) arriving before \(e\). So we have \(\phi'(u) + \phi'(v) \geq \frac{1}{2} \sum_{e' \in P(e)} \Delta\phi(e')\). By construction (Line 5) of Algorithm 2, we have

\[
w(e) = w'(e) + \phi'(u) + \phi'(v) \geq \frac{1}{2}\Delta\phi(e) + \frac{1}{2} \sum_{e' \in P(e)} \Delta\phi(e') = \frac{1}{2} \sum_{e' \in P(e)} \Delta\phi(e').
\]

**Lemma 4.** The matching \(M\) produced by Algorithm 2 is \(\frac{1}{2(1+\varepsilon)}\)-approximate.

**Proof.** We denote the constant in the algorithm by \(\varepsilon\). We observe that each edge \(e \in E_S\), which does not belong to the matching \(M\), has a neighboring edge in \(M\) inspected later than \(e\) in the stream. Hence

\[
\sum_{e \in M} w(e) \geq \frac{1}{2} \sum_{e \in M} \sum_{e' \in P(e)} \Delta\phi(e') \geq \frac{1}{2} \sum_{e \in E_S} \Delta\phi(e) = \frac{1}{2} \sum_{e \in E} \Delta\phi(e) = \frac{1}{2} \sum_{v \in V} \phi(v) \quad [\Delta\phi(e) = 0 \text{ if } e \notin E_S]
\]

\[
\geq \frac{1}{2(1 + \varepsilon)} \sum_{v \in V} \phi(v) \geq \frac{1}{2(1 + \varepsilon)} \sum_{e \in M^*} w(e) \quad [\text{from Obs. 1 and Cor. 2}].
\]
Algorithm 3 Finding a locally dominant edge.

```
1: function bestMate(u, S[u])
2:     bestU = -1
3:     while S[u] is not empty do
4:         if S[u].top() is marked as deleted then
5:             S[u].pop()
6:             continue
7:     bestU = S[u].top()
8:     break
9:     return bestU

10: procedure LdCheck(u, S, M)
11:     if u is not marked as deleted then
12:         v = bestMate(u, S[u])
13:         if v = -1 then
14:             Mark u as deleted; return
15:         bestV = bestMate(v, S[v])
16:         if u = bestV then ▷ Locally dominant
17:             mate[u] = v; mate[v] = u
18:             S[u].pop(); S[v].pop()
19:             Mark u and v as deleted
```

Input: a vertex u; the stacks S; Current Matching M
Output: Matching endpoints for each vertex, mate

Note that the $\varepsilon = 2\varepsilon'$ here is yet another constant. We could also express the approximation ratio as $\frac{1}{2} - \varepsilon$ as $\frac{1}{2}\varepsilon'$ is equivalent to it. The memory requirement of the new algorithm is the same as that of the serial one; since both these algorithms retain the same edges, and the new post-processing requires memory linear in the size of $E_S$.

4.2 Parallel Implementation

We have proved that any algorithm that computes a locally dominant matching is $\frac{1}{2} - \varepsilon$-approximate, and now we discuss the detailed implementation of our locally dominant algorithm that builds on a Procedure LdCheck described next. In LdCheck, for a vertex u, we find its best choice for mate, say v, in the order of its stack and check whether u is also the best choice for v. In that case, $\{u, v\}$ is a locally dominant edge. We apply the matching operation on this edge, which includes popping both stacks, updating the mate arrays, and marking u and v as deleted.

To be efficient, we maintain a work queue that stores the candidate vertices for matching (a vertex may be stored multiple times). Initially, this queue holds all the vertices. The algorithm continues by deleting a vertex from the work queue until the queue is empty and applies the LdCheck procedure on this vertex. When an edge is matched, all of its available neighbors are inserted in the queue since, for these vertices, the corresponding stacks could potentially be modified. A vertex can be inserted into the queue at most $|N_{E_s}(v)|$ times, and the total amortized work of a vertex (w.r.t all of its insertions in the queue) is $O(|N_{E_s}(v)|)$. The overall time complexity is thus linear in the size of $E_S$ which is $O(n \log n)$.

In the shared memory parallel implementation, we maintain queues for each of the processors. Initially, all the queues form a partition of $V$. Each processor in parallel runs the LdCheck procedure on the vertices of its own queue and updates the queues similarly to the serial algorithm. In the parallel algorithm, a vertex can be in multiple queues and thus may call LdCheck simultaneously. Moreover, both endpoints of an edge can initiate LdCheck at the same time. To resolve these, we employ synchronization by defining a lock for each vertex. If an edge $\{u, v\}$ is locally dominant, instead of applying the matching operations immediately, we use the test-and-lock function to lock $x$, where $x = \min\{u, v\}$. If the lock test fails, then the vertex is ignored. Otherwise, we recheck the availability of $u$ and $v$, if
perform the pop operation, update the matching, and mark \( u \) and \( v \) as deleted. The lock is released afterward. The detailed pseudocode of \texttt{LdCheck} is in Algorithm 3 with the critical region highlighted.

The total work of the parallel algorithm is the same as the serial, i.e., linear in the size of edges in the stack. (The depth might not be \( \mathcal{O}(\log n) \) even when the edge weights in the graph are random or the edges are streamed in random order since the stacks keep a non-random subset of edges). By induction on the edges matched by the post-processing algorithm of Paz and Schwartzman, we can show that the matchings obtained by the two algorithms are identical.

We implemented our parallel post-processing using the OpenMP library. Since the number of edges in the post-processing step is relatively small (geometric mean of the ratio of the stack size to the number of edges is only 2.65% for LARGE graphs), the added overhead of the parallel algorithm provides only slight benefits over the original post-processing, even when 128 processors are used. However, we expect that for massive graphs with trillions of edges, our parallel implementation could outperform the sequential one significantly.

## 5 Semi-streaming MWC Algorithms

### 5.1 Nearest Neighbor (NN) Algorithm

The 2-approximate Nearest Neighbor (NN) algorithm chooses an edge of minimum weight incident on a vertex to cover it. The NN algorithm can be implemented easily in a semi-streaming computational model. We maintain an array of edges (\( \mu \)) of size \( n \), representing a minimum weighted edge incident on a vertex, initialized to the empty set and updated in the course of the algorithm. When an edge \( \{u, v\} \) arrives in the stream, we compare its weight with the weights of \( \mu(u) \) and \( \mu(v) \) and update them accordingly.

\begin{lemma}
The approximation ratio of the NN algorithm is 2, and its memory requirement is \( \mathcal{O}(n \log n) \) bits.
\end{lemma}

\begin{proof}
Let \( C^* \) be an optimum edge cover, and let \( C \) denote an edge cover computed by the NN algorithm. For any vertex \( v \in V \), we denote by \( \nu(v) \) an arbitrary edge in \( N_{C^*}(v) \). Since \( \mu(v) \) is a minimum weighted edge incident on \( v \), we have \( w(\nu(v)) \geq w(\mu(v)) \). An edge in the optimal solution can cover at most two vertices. Hence \( w(C^*) \geq \frac{1}{2} \sum_{v \in V} w(\nu(v)) \geq \frac{1}{2} \sum_{v \in V} w(\mu(v)) = \frac{1}{2} w(C) \).

For each vertex \( v \), NN stores \( \mu(v) \), which requires \( \mathcal{O}(\log n) \) bits, and hence the memory requirement.
\end{proof}

### 5.2 Two pass \( \frac{3}{2} + \varepsilon \)-approx (TwoPass) Algorithm

We will exploit a relationship between a minimum weighted edge cover (MWC) and a maximum weighted matching (MWM) to design a two-pass streaming algorithm. An MWC can be constructed by solving an MWM on a weight-transformed graph. For a vertex subset \( W \), we will denote by \( \mu(W) \) the set of minimum weight edges incident on vertices in \( W \). We denote by \( V(M) \) the set of endpoints of edges in a matching \( M \). The following observation shows that some edges of the graph can be pruned for the MWC problem.

\begin{observation}
Any edge \( e \in E \) with weight \( w(e) \) such that \( w(e) \geq w(\mu(u)) + w(\mu(v)) \) can be removed without changing the weight of an optimal edge cover.
\end{observation}
We define the transformed weight of an edge \( \{u, v\} \) as \( w'(\{u, v\}) = w(\mu(u)) + w(\mu(v)) - w(\{u, v\}) \). By Observation 6, we may discard any edges with \( w'(\{u, v\}) \leq 0 \). Let \( E' \) be the set of edges in \( G \) after removing such edges, and \( G' = (V, E', w') \) be the subgraph induced by \( E' \). Let \( M^* \) be a maximum weight matching in \( G' \). Then an edge cover can be constructed as \( C^* = M^* \cup \mu(V \setminus V(M^*)) \). We show in Lemma 7 that \( C^* \) is a minimum weight edge cover.

\[ \text{Lemma 7. Let } M^* \text{ be a maximum weight matching obtained with the transformed weights } w' \text{ from a graph } G(V, E, w). \text{ Then the edges in } M^* \text{ together with a set of minimum weight edges incident on the unmatched vertices constitute a minimum weight edge cover } C^* \text{ of the graph } G. \]

**Proof.** By construction of \( C^* \), we have

\[
w(C^*) = w(M^*) + w(\mu(V \setminus V(M^*))) \leq \sum_{e=\{u,v\}\in M^*} w(\mu(u)) + w(\mu(v)) - w'(e) + w(\mu(V \setminus V(M^*))) = w(\mu(V)) - w'(M^*).
\]

Let \( C \) be an arbitrary edge cover, and \( M \subseteq C \) be an arbitrary maximal matching (a matching to which we cannot add an edge without violating the matching conditions) in \( G' \). We construct a new edge cover, \( C' \), with a possibly lower weight than \( C \) by \( C' = M \cup \mu(V \setminus V(M)) \). Since each of the edges in \( C \setminus M \) covers a single vertex in \( V \setminus V(M) \), we have \( w(C') \leq w(C) \). It follows that \( w(C) \geq w(C') = w(\mu(V)) - w'(M) \). This shows the equivalence of \( \text{MWM} \) and \( \text{MWC} \). Since \( w(\mu(V)) \) is a constant, \( w(C) \) is minimized when \( w'(M) \) is maximized.

In fact, this transformation is also approximation preserving. We state and prove the following theorem from [24].

\[ \text{Theorem 8. Let } M \text{ be a } (1 - \alpha)\text{-approximate matching obtained with the transformed weights } w' \text{ from a graph } G = (V, E, w). \text{ Then the edges in } M \text{ together with a set of minimum weight edges incident on the unmatched vertices constitute a } (1 + \alpha)\text{-approximate edge cover } C \text{ of the graph } G \text{ with respect to the original weights } w. \]

**Proof.** Let \( C^* \) denote an optimal edge cover with respect to the weights \( w \) and let \( M^* \) denote an optimal matching with respect to the weights \( w' \). It is easy to verify that \( w'(\{u, v\}) \leq w(\{u, v\}) \) for all edges \( \{u, v\} \). We have

\[
w'(M^*) = \sum_{\{u, v\} \in M^*} w'(\{u, v\}) \leq \sum_{\{u, v\} \in M^*} w(\{u, v\}) = w(M^*) \leq w(C^*). \tag{3}
\]

From the construction of \( C \) and the definition of \( M \), we have

\[
w(C) = w(M) + \mu(V \setminus V(M)) = \mu(V(M)) - w'(M) + \mu(V \setminus V(M)) = \mu(V) - w'(M).
\]

Using the approximation ratio of the matching algorithm that computed \( M \), we obtain

\[
w(C) = \mu(V) - w'(M) \leq \mu(V) - (1 - \alpha)w'(M^*) = \mu(V) - w'(M^*) + \alpha w'(M^*) = w(C^*) + \alpha w'(M^*) \leq (1 + \alpha) w(C^*).
\]

In the second line, first we use the equation in the proof of Lemma 7, and next we use inequality (3). This completes the proof.
We develop here another one-pass (TwoPass) streaming edge cover algorithm. In the first pass, we find the edges \( \{ \mu(v) : v \in V \} \) using the NN Algorithm discussed in Section 5.1. In the next pass, we construct the reduced weight for an edge in the stream and employ the PS streaming matching algorithm discussed in Section 4. Any vertices \( v \) that are unmatched in the matching are then covered by the edges from \( \{ \mu(v) \} \). Since the PS algorithm is \( \left( \frac{1}{2} - \frac{\epsilon}{ \log n} \right) \) \( \geq \frac{1}{2} - \frac{\epsilon}{ \log n} \)-approximate, by Theorem 8, the TwoPass algorithm is \( (\frac{3}{2} + \epsilon) \)-approximate for a constant \( \epsilon \geq 0 \). This algorithm requires an additional \( \mathcal{O}(n \log n) \) bits space, to store the \( \mu(.) \)s, compared to the PS algorithm.

5.3 One pass 2-approx (OnePass) Algorithm

We develop here another one-pass 2-approximation algorithm for the \( \text{MWC} \) problem. Although this OnePass algorithm provides the same worst-case approximation guarantee and has the same memory requirement as NN, empirically it obtains lower weights.

For each vertex \( v \in V \), the algorithm maintains a potential function \( \phi(v) \) initialized to \( \infty \) and a \( \text{tag}(v) \) initialized to zero. It also maintains an array \( \text{cover} \) for each vertex \( v \in V \) that stores the covering edge of \( v \). We say an edge is a 2-covering edge if half of its weight is smaller than the current potentials of both of its endpoints. It is a 1-covering edge if the weight of this edge is smaller than the potential of only one of the endpoints. When an edge \( e \{u,v\} \) arrives in the stream, the algorithm updates \( \mu(u) \) and \( \mu(v) \), if necessary. Then it checks whether it is a 2-covering or 1-covering edge, and processes it as described below. If the edge satisfies neither condition, it discards the edge.

- **\( e \) is a 2-covering edge.** We update \( \phi(u) \) and \( \phi(v) \) values to \( w(e)/2 \), and tag \( u \) and \( v \) as vertices covered by a 2-covering edge by assigning \( \text{tag}(u) = \text{tag}(v) = 2 \). The algorithm also updates the values of \( \text{cover}(u) \) and \( \text{cover}(v) \) to \( e \). If \( u \) or \( v \) was covered earlier by a 2-covering edge (Line 7 is true), we change the potential of the other endpoint, say \( y \), to \( w(\mu(y)) \). We also set \( \text{cover}(y) = \mu(y) \), and tag \( y \) to be covered by a 1-covering edge.

- **\( e \) is a 1-covering edge.** Denote the covering endpoint by \( u \). We update \( \text{cover}(u) = \mu(u) \), the potential \( \phi(u) = w(\mu(u)) \), and set tag \( u \) to 1.

When the streaming phase ends, the \( \mu(.) \) array stores a minimum weight edge incident on each vertex. In the post-processing phase, for each vertex \( v \), we update \( \text{cover}(v) = \mu(v) \), and \( \phi(v) = w(\mu(v)) \) if \( v \) is marked as covered by a 1-covering edge. Finally, \( C = \bigcup_{v \in V} \text{cover}(v) \) is returned as the edge cover. We show the detailed pseudocode in Algorithm 4.

To prove the approximation guarantee, we start with a simple observation.

\[ \textbf{Observation 9.} \ w(C) = \sum_{v \in V} \phi(v). \]

\[ \textbf{Lemma 10.} \text{ After the post processing phase, } \phi(u) \leq w(\mu(u)) \text{ for all } u \in V. \]

**Proof.** If \( u \) is covered by a 1-covering edge, then after the post-processing step, \( \phi(u) = w(\mu(u)) \). Hence assume that \( u \) is covered by a 2-covering edge, say \( e \{u,v\} \). There are two cases to consider. i) When the edge \( \mu(u) \) streams before \( \{u,v\} \). Since \( \mu(u) \) was not considered to cover \( u \), we have \( \phi(u) \leq w(\mu(u)) \). ii) When the edge \( \mu(u) \) is streamed before \( \{u,v\} \). Consider the iteration after \( \mu(u) \) arrives and before \( u \) is covered by \( \{u,v\} \). If in any one of these iterations, \( u \) is covered by a 1-covering edge, \( \phi(u) \) becomes \( w(\mu(u)) \). When \( \{u,v\} \) arrives, \( \phi(u) \) must be less than \( w(\mu(u)) \). On the other hand, if \( u \) is always covered by a 2-covering edge in all these iterations, then trivially \( \phi(u) \leq w(\mu(u)) \).

\[ \textbf{Lemma 11.} \text{ The OnePass algorithm is 2-approximate and uses } \mathcal{O}(n \log n) \text{ bits of memory.} \]
Algorithm 4 OnePass Edge Cover({e(u, v)}).

Input: Stream of edges e(u, v)
Output: 2-approximate Edge Cover C, using O(n log n) bits space
1: ∀v ∈ V : φ(v) = ∞, tag(v) = 0, cover(v) = ∅, µ(v) = ∅; C ← ∅ ▷ Initialization
2: ▷ Stream Process
3: for e{u, v} ∈ E do
4: Update µ(u) and µ(v)
5: if e is a 2-covering edge then
6: for x ∈ {u, v} do
7: if tag(x) = 2 then ▷ y is the other endpoint of the edge that covers x
8: y = cover(x) \ x
9: Update cover(y), tag(y) and φ(y)
10: Update cover(x), tag(x) and φ(x)
11: else if e is a 1-covering edge of x then
12: Update cover(x), tag(x) and φ(x)
13: ▷ Post Processing
14: if tag(v) = 1 then
15: Update φ(v) and cover(v) ▷ φ(v) is updated for analysis only
16: C = ⋃v∈V cover(v)

Proof. Let C* be an optimal edge cover. By Observation 9,

\[ w(C) = \sum_{u \in V} \phi(u) \leq \sum_{u \in V} w(\mu(u)) \leq 2w(C^*). \]

The last two inequalities follow from Lemma 10 and the proof of Lemma 5, respectively. ▷

For each vertex v ∈ V, the OnePass algorithm stores the edges µ(v), cover(v), and a flag value representing whether this vertex is covered by 2-covering or 1-covering edge. Thus, the total memory per vertex is O(log n) bits, and the memory required is O(n log n) bits.

6 Experimental Results

We have implemented the streaming matching and edge cover algorithms discussed in the previous sections and performed a thorough comparison of the algorithms with a number of offline approximation algorithms that are implemented on a node of a cluster computer to meet the memory requirements of the large graphs we work with. For larger graphs, the shared memory parallel (OpenMP) version of the ROMA algorithm (Table 2) is used, since it is compute-intensive. All other algorithms are sequential. The algorithms are executed on a node of a community cluster computer with 128 cores in the node, where the node is an AMD EPYC 7662 with 1 TB of total memory. The algorithms are implemented in C++17 and compiled with g++9.3.0 using the -O3 optimization flag.

We provide two options for streaming of the edges: one option directly reads the edges from a file one by one, which we call TrueStream, while the other reads all the edges into memory first and then simulates the streaming algorithm by reading edges from memory, which we call SimStream. The TrueStream setting is used for comparing the overall runtime and memory of the competing algorithms and the SimStream is used for comparing the actual processing time of the algorithms without the graph reading and construction time. The SimStream also reveals the performance of the streaming algorithms executing in an offline setting. For reporting memory, we use the getrusage system call to retrieve the maximum resident set size (RSS) during the program’s execution. We refer to the title page for the source code repository that contains our implementation of the streaming algorithms.
Table 1 Graph statistics of our dataset. K, M and B stand for thousand, million and billion, respectively.

(a) Small graph instances. All the graphs are weighted.

<table>
<thead>
<tr>
<th>Graph</th>
<th>n</th>
<th>m</th>
<th>Avg. Deg.</th>
<th>Max. Deg.</th>
<th>Min. Deg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>astro-ph</td>
<td>16,706</td>
<td>121,251</td>
<td>14.52</td>
<td>360</td>
<td>0</td>
</tr>
<tr>
<td>Reuters911</td>
<td>13,332</td>
<td>148,038</td>
<td>22.21</td>
<td>2,265</td>
<td>0</td>
</tr>
<tr>
<td>cond-mat-2005</td>
<td>40,421</td>
<td>175,691</td>
<td>8.69</td>
<td>278</td>
<td>0</td>
</tr>
<tr>
<td>gas_sensor</td>
<td>66,917</td>
<td>818,224</td>
<td>24.45</td>
<td>32</td>
<td>7</td>
</tr>
<tr>
<td>turom_m</td>
<td>189,924</td>
<td>778,531</td>
<td>8.20</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Fault_639</td>
<td>638,802</td>
<td>13,303,571</td>
<td>41.65</td>
<td>266</td>
<td>0</td>
</tr>
<tr>
<td>mouse_gene</td>
<td>45,101</td>
<td>14,461,095</td>
<td>641.27</td>
<td>8,031</td>
<td>0</td>
</tr>
<tr>
<td>bone010</td>
<td>986,703</td>
<td>23,432,540</td>
<td>47.50</td>
<td>62</td>
<td>11</td>
</tr>
<tr>
<td>dielFil.V3real</td>
<td>1,102,824</td>
<td>44,101,598</td>
<td>79.98</td>
<td>269</td>
<td>8</td>
</tr>
<tr>
<td>kron.logn21</td>
<td>2,097,152</td>
<td>91,040,932</td>
<td>79.98</td>
<td>213,904</td>
<td>0</td>
</tr>
</tbody>
</table>

(b) Large graph instances. U: Unweighted, W: Weighted Graph.

<table>
<thead>
<tr>
<th>Graph</th>
<th>n</th>
<th>m</th>
<th>Avg. Deg.</th>
<th>Max. Deg.</th>
<th>Min. Deg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcielskian20 (U)</td>
<td>786.43</td>
<td>K</td>
<td>1.36</td>
<td>3446.42</td>
<td>393,215</td>
</tr>
<tr>
<td>com-Friendster (U)</td>
<td>65.61</td>
<td>M</td>
<td>1.81</td>
<td>55.06</td>
<td>5,214</td>
</tr>
<tr>
<td>GAP-kron (W)</td>
<td>134.22</td>
<td>M</td>
<td>2.11</td>
<td>31.47</td>
<td>1,572,838</td>
</tr>
<tr>
<td>GAP-urand (W)</td>
<td>134.22</td>
<td>M</td>
<td>2.15</td>
<td>32</td>
<td>68</td>
</tr>
<tr>
<td>MOLIERE_2016 (W)</td>
<td>30.24</td>
<td>M</td>
<td>3.34</td>
<td>220.81</td>
<td>2,106,904</td>
</tr>
<tr>
<td>Agatha_2015 (U)</td>
<td>183.96</td>
<td>M</td>
<td>5.79</td>
<td>62.99</td>
<td>12,642,631</td>
</tr>
</tbody>
</table>

(c) ML Datasets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th># of items</th>
<th># of features</th>
<th># of edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMDb</td>
<td>45,039</td>
<td>30,000</td>
<td>992.94 M</td>
</tr>
<tr>
<td>MNIST</td>
<td>60,000</td>
<td>784</td>
<td>1.80 B</td>
</tr>
<tr>
<td>DBpedia</td>
<td>545,721</td>
<td>10,000</td>
<td>40.42 B</td>
</tr>
</tbody>
</table>

6.1 Dataset and Benchmark Algorithms

Our testbed (shown in Table 1) consists of three datasets. In Table 1a, we show the SMALL dataset, which includes ten weighted graphs from the Suitesparse Matrix Collection [7] frequently used for comparing matching algorithms [28, 20]. In Table 1b, we show the LARGE benchmark data that consists of six of the largest undirected graphs in the Suitesparse Matrix Collection [7], with each graph having more than 1 billion edges. We assign uniform random weights in the range [1, 10^6] for the three unweighted graphs. Table 1c lists three popular machine learning datasets. The MNIST is a hand-written digit recognition dataset, while the two others are text classification data. IMDb [31] is a binary sentiment analysis dataset, and DBpedia [30, 43] is ontology classification data.

We compare the streaming algorithms with a few representative approximate offline algorithms from the literature listed in Table 2. These algorithms are practical to implement [32] and have a similar approximation guarantee as the streaming ones. The parameter ε in the PS algorithm is set to 10^{-3} unless otherwise specified. Following [32], we ran only one phase of ROMA (n random augmentations) on an initial matching generated by GPA algorithm (GPA+ROMA). The PGA, GPA and parallel ROMA implementations are due to Manne and Berge [5]. For a detailed description of these approximation algorithms, we refer the reader to [32, 39].
Table 2 Benchmark Approximation Offline Algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Approx. Ratio</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greedy [3]</td>
<td>1/2</td>
<td>$\mathcal{O}(m \log n)$</td>
</tr>
<tr>
<td>Global Path Algo. (GPA) [32]</td>
<td>1/2</td>
<td>$\mathcal{O}(m \log n)$</td>
</tr>
<tr>
<td>Path Growing Algo. (PGA) [8]</td>
<td>1/2</td>
<td>$\mathcal{O}(m)$</td>
</tr>
<tr>
<td>Random Order Augmentation Matching Algo. (ROMA) [37]</td>
<td>2/3 - $\varepsilon$</td>
<td>$\mathcal{O}(m \log \frac{1}{\varepsilon})$</td>
</tr>
<tr>
<td>Primal-dual \text{MW}C (PD) [17]</td>
<td>3/2</td>
<td>$\mathcal{O}(mn)$</td>
</tr>
</tbody>
</table>

6.2 Matching Results

Figure 2 shows the relative weight, memory, and runtime comparisons of the streaming and offline matching algorithms for the SMALL (Figure 2a) and LARGE (Figure 2b) datasets. The relative quantities are computed w.r.t. a baseline algorithm. For weights, the baseline is GPA+ROMA, while for the runtime and memory it is the FB algorithm. The number of runs for each graph and algorithm pair is set to five for the SMALL dataset and three for the LARGE graphs. The relative quantity of interest is obtained by computing the mean value of the quantity over the runs and then taking its ratio with the mean baseline value.

The top-left plots in both of the subfigures show the box plots of the weights achieved by the algorithms across all the graphs relative to the weights of GPA+ROMA. For reporting relative runtime and memory results we show line plots, where the $x$-axis represents the graph instances (sorted from high to low w.r.t. to the edges) and the $y$-axis represents the relative memories (bottom-left) or runtimes (top and bottom right) of the algorithms compared. The two right plots in both Subfigures in Figure 2 report the relative runtimes. Of these, the top right ones show the total runtimes that include reading, graph constructions, initialization, processing, and post-processing time. The bottom-right plots report only the algorithmic time excluding reading and graph construction time. For this experiment (bottom-right plots), when streaming algorithms are executed, we used the SimStream setting. For all other experiments, we used TrueStream streaming option. For the largest instance in our data set, the Agatha_2015 graph, the parallel ROMA did not terminate within three hours of runtime.

For this instance, we chose GPA weight to be the baseline. We report the quantitative results of the baseline algorithms for our dataset in Table 3.

Quality. From the top left plots in Figure 2, we see that the GPA+ROMA obtains higher weights than other algorithms on almost all the graphs. On the SMALL dataset, GPA and Greedy perform the next best with similar median relative weights. Then we have the PGA and the streaming algorithm PS. All these algorithms achieved relative weight $\geq 95\%$ of GPA+ROMA. The FB algorithm performs the worst achieving around 90\% of the relative weights in terms of median. On the LARGE dataset, the weight differences from the GPA+ROMA are more pronounced. In terms of median relative weights, the streaming PS and the GPA perform similarly with around 90\% in the median. The next are the Greedy and PGA. This result suggests that, for our benchmark instances, the algorithms follow their worst case approximation guarantee in a relative manner. The GPA+ROMA is $2/3 - \varepsilon$-approximate, and it achieves the best weight, while all the other offline algorithms are $1/2$-approximate. The quality of PS is better than the FB and comparable to the other half approximate offline algorithms. We note that for a few graphs (e.g., myc20), the streaming algorithms perform significantly worse than the others.
Memory. For memory usage (bottom left plots), as expected, we see that the streaming algorithms significantly outperform the offline ones. For both the datasets, the FB algorithm requires the smallest memory while the PS follows it closely. The Greedy algorithm is the next, which requires approximately twice as much memory than FB across the SMALL graphs. For LARGE graphs, the range is $[16, 2048] \times$. All other offline algorithms require approximately twice the memory of the Greedy. This experiment suggests the streaming algorithms are extremely memory efficient and well-suited to compute matchings in massive graphs.

Runtimes. Finally, we discuss the runtime results. All these algorithms, except the Greedy, GPA and GPA+ROMA, have linear time complexity. In terms of total runtime in the SMALL dataset (top-right plot in Figure 2a), we see that all the algorithms except GPA and GPA+ROMA are comparable, with PS completing faster for a few of the instances than FB. The GPA and GPA+ROMA require almost twice the time of FB for most of the instances, except for kron, where the GPA+ROMA is $8 \times$ slower. For the LARGE dataset, the streaming and the Greedy algorithms are comparable, except for the mv20 graph, where Greedy requires twice the time of the streaming algorithms. When we look into the algorithmic times of these competing algorithms (bottom-right plots in Figure 2, we see clear separations. FB is the fastest, and PS requires at most twice as much time as FB in most of the instances for SMALL graphs, while for LARGE graphs both streaming algorithms are similar. Next is the PGA requiring $2 \times$ to $4 \times$ more time than the FB for both SMALL and LARGE datasets. The Greedy algorithm is $2 \times$ to $16 \times$ slower than the FB on the SMALL dataset, while on LARGE ones, it is up to $32 \times$ slower. GPA and GPA+ROMA are the two slowest algorithms, where for the SMALL dataset the ranges of relative runtimes are $[4, 32]$ and $[32, 128]$, respectively. For the LARGE graph experiments, GPA needs $8 \times$ to $32 \times$ more time to complete than the baseline, whereas for GPA+ROMA the range is $[8, 64] \times$. We note that for the large graphs, the ROMA timing reported is from the parallel algorithm using 128 threads, and it did not finish for the Agatha_2015 graph in 3 hours.

These experiments suggest that the streaming algorithms are suitable for solving massive graph problems since they are more memory efficient and faster than the offline ones. We highlight the PS algorithm to be the overall winner since it often achieves quality close to the best weights while significantly outperforming the offline algorithms for memory and runtimes.

Effect of $\varepsilon$ on PS. Figure 1 shows the effect of the parameter $\varepsilon$ on the streaming matching algorithm, PS. We choose $\varepsilon$ from the set $\{0, 2^{-x} \}$, where $x = \{16, 14, 12, 10, 8, 6, 4, 2, 1, 0\}$. The value of $\varepsilon$ influences both the approximation ratio and the memory requirement of the algorithm. With increasing $\varepsilon$, we see that both the memory requirement and the weight of the matching decrease. But in almost all cases, the change in the weight is smaller than the decrease in the memory. This experiment suggests that we can substantially decrease memory without significantly decreasing the matching weight.

6.3 Edgecover Results

We now explain Figure 3, which shows the edge cover results for our streaming algorithms (NN, OnePass and TwoPass) and the offline primal-dual algorithm (PD). The experimental setup, number of runs, and subplot description of the figure are exactly the same as the matching experiments described in Section 6.2. For weights, we choose PD to be the baseline, while the runtime and memory results are relative to the NN algorithm.
Figure 1 Memory and weight changes of the streaming matching with varying \( \varepsilon \) on the LARGE dataset. The relative weights and memory are wrt the weights and memory using \( \varepsilon = 0 \).

**Quality.** The two top-left plots of Figure 3 show the relative weights of the edge cover algorithms. Here, the lower is the better. We see that on both SMALL and LARGE datasets, our TwoPass algorithm finds edge cover with smaller weights than the offline PD algorithm. The OnePass is better than NN algorithm in terms of median weight for both datasets, while the NN is the worst in terms of quality.

**Memory.** The bottom-left plots in the two subfigures of Figure 3 present the relative memory of the edge cover algorithms. The NN requires the least amount of memory space to execute, while both the OnePass and TwoPass need approximately twice the memory across all the datasets. An exception is the mouse_gene dataset, where the TwoPass algorithm takes 8× memory than the NN. The offline PD algorithm is significantly more memory-demanding than the streaming ones. For SMALL dataset, the PD algorithm requires 8-512× more memory, while for the LARGE dataset, the range is [16, 8192].

**Runtime.** The two right plots on the two subfigures of Figure 3 show the relative total runtime and algorithmic time taken by the algorithms. NN and OnePass behave similarly for both datasets, whereas TwoPass is about twice slower. TwoPass reads the graph twice from the file, which dominates the overall runtime. The offline PD is faster than the TwoPass algorithm and comparable to the NN. If we consider only the algorithmic time, the ranking order of the algorithms is: NN, OnePass, TwoPass and the offline PD for both SMALL and LARGE datasets.

These experiments highlight the applicability of our streaming algorithm to solve large-scale MWC problems. The geometric means of the relative quantities of MWM and MWC algorithms on both datasets are listed in Table 4.

### 6.4 Baseline Results

Table 3 presents the weight, time and memory quantities for the baseline algorithms for our experiment. Table 4 shows the geometric means of all the algorithms for both MWM and MWC across the SMALL and LARGE datasets. We can conclude that the streaming algorithms reduce the memory required by the offline algorithms by one or two factors of ten. They
are also faster than the offline algorithms, and the weights computed by the PS matching algorithm and the TwoPass edge cover algorithms are close to the values obtained by the best performing offline algorithms.

Table 3 Baseline algorithm results for matching and edge cover.

<table>
<thead>
<tr>
<th>Graph</th>
<th>MWM</th>
<th>MWC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time(s)</td>
<td>Proc. Mem</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>FB</td>
</tr>
<tr>
<td>Base. Alg.</td>
<td>GPA+ FB</td>
<td>GPA+ FB</td>
</tr>
<tr>
<td>astro-ph</td>
<td>6.44e+3</td>
<td>0.056</td>
</tr>
<tr>
<td>Reuters911</td>
<td>2.53e+4</td>
<td>0.050</td>
</tr>
<tr>
<td>cond-mat-2005</td>
<td>1.94e+4</td>
<td>0.086</td>
</tr>
<tr>
<td>turing_m</td>
<td>5.84e+4</td>
<td>0.547</td>
</tr>
<tr>
<td>gas_sensor</td>
<td>1.50e+3</td>
<td>0.925</td>
</tr>
<tr>
<td>Fault_039</td>
<td>2.57e+18</td>
<td>6.730</td>
</tr>
<tr>
<td>mouse_gene</td>
<td>1.33e+3</td>
<td>7.719</td>
</tr>
<tr>
<td>house900</td>
<td>1.51e+3</td>
<td>15.555</td>
</tr>
<tr>
<td>dieFilterV3real</td>
<td>1.76e+5</td>
<td>27.512</td>
</tr>
<tr>
<td>kron_g500-logn21</td>
<td>4.63e+6</td>
<td>30.595</td>
</tr>
</tbody>
</table>

Table 4 Geometric mean of quantities relative to baseline algorithms for the MWM and MWC problems. For baseline algorithms, see Table 3.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>SMALL Dataset</th>
<th>LARGE Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>MWM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FB</td>
<td>0.89</td>
<td>1.00</td>
</tr>
<tr>
<td>PS</td>
<td>0.96</td>
<td>21.22</td>
</tr>
<tr>
<td>Greedy</td>
<td>0.96</td>
<td>61.63</td>
</tr>
<tr>
<td>PGA</td>
<td>0.98</td>
<td>66.83</td>
</tr>
<tr>
<td>GPA+ROMA</td>
<td>1.00</td>
<td>66.74</td>
</tr>
<tr>
<td>MWC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NN</td>
<td>1.26</td>
<td>1.00</td>
</tr>
<tr>
<td>OnePass</td>
<td>1.20</td>
<td>2.17</td>
</tr>
<tr>
<td>TwoPass</td>
<td>0.97</td>
<td>2.57</td>
</tr>
<tr>
<td>PD</td>
<td>1.00</td>
<td>34.55</td>
</tr>
</tbody>
</table>

6.5 Matching on ML Dataset

Next we show results on graphs generated from Machine Learning datasets. To compute an item intersection graph from a dataset, for each distinct pair of items, we create an edge with an edge weight calculated from the two feature vectors of the items. Note that the item intersection graph is dense; indeed, it could be a complete graph. For our experiment, we have used cosine similarity for the IMDb and DBpedia datasets since they are text data. We compute edge weights for the MNIST by subtracting the squared Euclidean distance of the feature vectors from the maximum possible squared distance. Since the size of each image is $28 \times 28$ pixels (the size of the feature vector is 784), with pixel value in $[0,255]$, the maximum possible squared distance for MNIST is $784 \times 255^2$. The streaming algorithm assumes these edges are generated one by one and passed to the algorithm. But for the offline algorithms, we are required to generate all the edges and then apply the algorithm to them. We compare the PS algorithm (streaming) and the Greedy algorithm (offline), the latter chosen since it is most memory-efficient among offline algorithms.
In Table 5, we observe that while Greedy gets better weights for two of the datasets (8% and 3% increase in weights, respectively), it requires two orders of magnitude memory than the Streaming algorithm. For the DBpedia dataset, where the item intersection graphs contain more than 40 billion edges, the Greedy algorithm failed to terminate since it needed more than 1 TB of memory. This experiment also shows the utility of the streaming algorithm (particularly the PS algorithm) for dense graphs. The edge-retention ratio, defined as the ratio of the number of edges in the stack to the total number of edges (in percent), is extremely low, as shown in Table 5.

### 7 Conclusion and Future work

The best streaming algorithms for matching and edge cover compute weights close to the offline algorithms, but as expected, use one to two factors of ten less memory to solve the problems. In a machine learning application, we show that the offline algorithm can run out of memory where the streaming algorithm does not. The streaming algorithms are generally also faster than the offline algorithms. For the MWC, we describe a two-pass $\frac{3}{2} + \varepsilon$-algorithm, and an open question is whether there exists a single pass algorithm with the same guarantee.
Figure 2 Comparison of matching algorithms in the small and large dataset. $\epsilon = 0.001$ for the PS algorithm. Weight baseline: GPA+ROMA. Runtime and memory baseline: FB. For runtime and memory results, the y-axis scale is logarithmic.
(a) Offline and Streaming algorithm comparison for edge cover on the small dataset.

(b) Offline and Streaming algorithm comparison for edge-cover on the large dataset.

Figure 3 Comparison of edge cover algorithms in the small and large datasets. $\varepsilon = 0.001$ for the TwoPass algorithm. Weight baseline: PD, Runtime, and memory baseline: NN. For runtime and memory results, the y-axis scale is logarithmic.
References


15. SM Ferdous. smferdous1/GraST. Software (visited on 13/05/2024). URL: https://github.com/smferdous1/GraST.


We present a fast and efficient 3/2 dual approximation algorithm for CPU/GPU scheduling under the objective of makespan minimization. In CPU/GPU scheduling tasks can be scheduled on two different architectures. When executed on the CPU, a task is moldable and can be assigned to multiple cores. The running time becomes a function in the assigned cores. On a GPU, the task is a classical job with a set processing time. Both settings have drawn recent independent scientific interest. For the moldable CPU scheduling, the current best known constant rate approximation is a 3/2 approximation algorithm [Wu et al. EJOR volume 306]. The best efficient algorithm for this setting is a 3/2+ε approximation [Mounie et al. SIAM '07] whereas GPU scheduling admits a 13/11 approximation [Coffman, Garey, Johnson SIAM'78]. We improve upon the current best known algorithms for CPU/GPU scheduling by Bleuse et al. by formulating a novel multidimensional multiple choice knapsack to allot tasks to either architecture and schedule them there with known algorithms. This yields an improved running time over the current state of the art. We complement our theoretical results with experimentation that shows a significant speedup by using practical optimizations and explore their efficacy.
executing the task this function is not linear in the amount of used processors. Both of these
settings have been studied in the context of makespan minimization [17, 8]. The makespan
corresponds to the earliest time at which all tasks in the given instance finish when all
processors start at the same point in time. Applications of makespan scheduling are again
widespread.

In this paper, we consider the CPU/GPU scheduling problem. Here, each task can be
executed on either the GPU or CPU architecture, i.e. it has both a processing time when
executed on the GPU and a function representing the moldable work for execution on the
CPU. We present a low complexity algorithm that yields a $\frac{3}{2} + \varepsilon$ approximation guarantee
of the maximum makespan.

1.1 Notation

We define the natural numbers as $\mathbb{N} := \{1, 2, 3, \ldots\}$ and $\mathbb{N}_0 := \mathbb{N} \cup \{0\}$, where $\cup$ denotes the disjoint union. More, we define $[n] := \{m \in \mathbb{N} : m \leq n\}$ and $[n]_0 := [n] \cup \{0\}$ for all $n \in \mathbb{N}_0$.

1.2 Problem Definition

In $P||C_{\text{max}}$ (or GPU scheduling) a task can be executed only sequentially on one machine.
An instance is given by a number $n \in \mathbb{N}$ of tasks, a number $m \in \mathbb{N}$ of machines and processing
times $p_j \in \mathbb{Q}_{>0}$ for all tasks $j \in [n]$. The goal is to find a schedule $\pi : [n] \to [m]$ of tasks to
machines, that minimizes the maximum makespan $\max_{i \in [m]} \sum_{j \in \pi^{-1}} p_j$.

In $P|\text{fct}|C_{\text{max}}$ (or CPU scheduling, also often called scheduling independent monotonic
moldable tasks) a task can be executed on multiple machines in parallel. An instance is
given by a number of tasks $n \in \mathbb{N}$, a number of machines $m \in \mathbb{N}$, and for each task $j \in [n]$ a function $p_j \in \mathbb{Q}_{>0}$, where $p_j(\ell)$ describes its processing time when scheduled on $\ell$
machines. We define the work (area) function $w_j : [m] \to \mathbb{Q}_{>0}$ for every task $j \in [n]$ by
$w_j(\ell) := \ell p_j(\ell)$. Also, we assume the monotony properties that first, $p_j$ is non-increasing
and second, $w_j$ is non-decreasing for all tasks $j \in [n]$. These represent the assumption that
assigning more machines to a task does not increase its processing time but may come with
an increase in work due to communication overhead. Further, we define for each task $j \in [n]$
the function for the canonical number of machines $\gamma_j : \mathbb{Q}_{>0} \to [m]$ such that $\gamma_j(h)$ is the
minimal number of machines needed to execute task $j$ in time $h$, and by convention set to
$+\infty$ if no such number exists.
A schedule is a mapping \( \pi : [n] \rightarrow \mathbb{Q}_{>0} \times \mathbb{N} \), that maps to each task a starting time and a number of machines. The maximum makespan of a schedule is \( \max \{ s + p_j(\ell) : j \in [n], (s, \ell) = \pi(j) \} \). A schedule is valid if the total number of machines that tasks are scheduled to does not exceed \( m \) at any point in time. The goal is to find a schedule that minimizes the maximum makespan. For an illustration of such a schedule see Figure 1(b).

CPU/GPU scheduling is a combination of CPU scheduling and GPU scheduling, where each task can be scheduled to either the CPU or the GPU. An instance consists of a number of tasks \( n \in \mathbb{N} \), the number of CPU machines \( m_c \) and the number of GPU machines \( m_g \), as well as a processing time on the GPU and a processing time function on the CPU for each task. A valid schedule is a partition of the tasks onto the CPU and GPU and a valid schedule on each for its respective partition. The maximum makespan is the maximum of both makespans. The goal is to find a schedule that minimizes the overall maximum makespan.

1.3 Related Work

CPU/GPU scheduling combines two scheduling paradigms. It is known that, on their own, both CPU and GPU scheduling are NP-complete [15, 11]. Furthermore, without monotony assumptions CPU-scheduling does not admit an approximation algorithm with an approximation rate less than \((3/2)\) unless \( P = NP \) [3].

For moldable CPU scheduling, there exist several approximation algorithms. Most relevant to this work is the algorithm by Mounié et al. [17]. They present a \((3/2 + \varepsilon)\)-approximation algorithm that runs in time \( O(n m_c \log(n/\varepsilon)) \). Their paper is in turn based on a two-phase approach by Turek et al. [18]. There have been several improvements made to the running time of the algorithm by Mounié. First, Jansen and Land reduced the dependence of the running time regarding \( m \) to a poly-logarithmic factor [10]. Later on, Grage et al. further improved this poly-logarithmic dependence, yielding a running time of \( O(n \log^2(\frac{1}{\varepsilon} + \log(m_c)) + \frac{n}{\varepsilon} \log^2(\frac{1}{\varepsilon}) \log(\varepsilon m_c)) \). The best approximation ratio for this problem is a \((3/2)\) approximation algorithm given by Chen et al. [19]. Furthermore, it is known that this problem admits a PTAS [12] however its running time depends on a superexponential function. Jansen and Land show that there exists an FPTAS for the case where there are many \((m > 8 \cdot n/\varepsilon)\) machines [10]. The running time of the FPTAS in this case was again improved in [6].

On the GPU side we model one of the most widespread and well-researched scheduling problems, makespan scheduling on parallel machines. This problem has been researched continuously since at least 1969 [7]. As the research in this field is so varied, we focus only on the results relevant to our work. For constant rate approximations, there exists the \((4/3)\) approximation LPT-algorithm [7], which runs efficiently in \( O(n \log(n)) \). Furthermore, there is the best known constant rate approximation algorithm provided by MULTIFIT, with a ratio of \((13/11)\) [13]. It is known that this problem admits an EPTAS [9]. More relevant to our result is the PTAS developed by Hochbaum and Shmoys [8]. They present the dual approximation framework, which we also employ to generate our solution.

The combination of both settings, i.e. CPU/GPU scheduling, was first considered by Bleuse et al. in [1]. They present an ILP based \((3/2 + \varepsilon)\)-approximation algorithm whose complexity is not polynomial in theory. Additionally, the authors claim that adapting a technique from [2] yields a dual approximation algorithm with running time \( O(n^2 m_c^2 m_g^2) \) and the same approximation ratio however, they do not explain how to adapt the technique.
1.4 Our Contribution

We present a low complexity \((3/2 + \varepsilon)\)-algorithm for CPU/GPU scheduling that runs in time \(O(\log(nm_c m_g/\varepsilon) \cdot n^2 m_c m_g)\) and optimize it for the case \(n > m^2_g\) such that it then runs in time \(O(\log(nm_c m_g/\varepsilon) \cdot nm_c \max(n, m^2_g))\). Our algorithm follows the idea and structure of the CPU scheduling algorithm by Mounié et al. [17] and extends it to the CPU/GPU setting. Further, we implemented our algorithms and evaluated them experimentally.

2 Description of the algorithm

In this section, we describe the algorithm that comprises our main result. We begin by describing the framework in which we embed the algorithm. Then we describe how to generate suitable bounds to apply the framework. Next, we present the main part of the algorithm, the allocation of tasks onto machines. We finish the description of the algorithm by describing how we calculate a feasible schedule for the sequential machines and argue the complexity of the entire algorithm in its framework.

2.1 Dual Approximation Framework

Our algorithm uses the dual approximation framework [8].

**Definition 1** (\(\lambda\)-dual approximation problem). Let \(\lambda > 1\). The \(\lambda\)-dual approximation problem to a makespan minimization scheduling problem consists of a scheduling instance \(I\) and a makespan guess \(d\). The goal is to either find a schedule with a makespan of at most \(\lambda d\) or decide that no schedule with a makespan at most \(d\) exists.

Now, given lower and upper bounds for the minimal makespan, one can use a \(\lambda\)-dual approximation algorithm to binary search for the minimal makespan and obtain a \(\lambda(1 + \varepsilon)\)-approximation for the primal problem. This yields the following lemma, a proof of which can be derived from [8].

**Lemma 2** (primal approximation). Let \(I\) be a makespan minimisation scheduling instance, \(d^*\) its minimal makespan, \(k \geq 0\), \(\lambda > 1\), \(\varepsilon > 0\), and \(\omega \in \mathbb{Q}_{>0}\) a lower bound such that \(\omega \leq d^* \leq (2 + k)\omega\). Then we can construct from a \(\lambda\)-dual approximation algorithm that runs in time \(O(f(I))\) a \(\lambda(1 + \varepsilon)\)-approximation algorithm for the primal problem that runs in time \(O(\log(1/\varepsilon + k) f(I))\).

The goal of this paper is to construct a \(3/2\)-dual approximation algorithm for CPU/GPU scheduling to then use Lemma 2. But first, we have to find an algorithm that constructs a suitable lower bound on the optimal makespan. For CPU scheduling, Ludwig and Tiwari [16] describe an algorithm that computes a lower bound in time \(O(mn \log n)\). Next, we show that their approach can be adapted to CPU/GPU scheduling.

**Lemma 3.** For an instance of CPU/GPU scheduling with optimal makespan \(d^*\), a lower bound \(\omega \in \mathbb{Q}_{>0}\) can be computed in time \(O((m_c n + m_g) \log(m_c n + m_g))\), with \(\omega \leq d^* \leq 4\omega\).

**Proof.** Let \(t_j : [m_c] \to \mathbb{Q}_{>0}\) be the processing time function, \(w_j\) the work area function on the CPU, and \(p_j\) the processing time on the GPU for each task \(j \in [n]\).

Now we define the mapping \(\omega : [m_c]^{[n]} \times \mathcal{P}([n]) \to \mathbb{Q}_{>0}\) by

\[
\omega(\alpha, A) := \max \left\{ \frac{1}{m_c} \sum_{j \in A} w_j(\alpha(j)), \max_{j \in A} \{t_j(\alpha(j))\}, \frac{1}{m_g} \sum_{j \in A^C} p_j, \max_{j \in A^C} \{p_j\} \right\}.
\]
yields a 2-approximation. Here, computing the term that assigns each task to the machine on which it takes the least time, is a 2-approximate NP-minimization for two identical machines is already in H. So one can find for the second machine to be assigned to the first machine, which can be modeled by setting their processing time and p

\[ m_1, m_2 \] for some \[ m \].

Next, we will derive an efficient way to compute \( \omega \).

Let \( H := \{ p_j : j \in [n] \} \cup \{ t_j(p) : j \in [n], p \in [m_c] \} \) be the set of processing times of tasks on the CPU/GPU. Further, let \( h_c(\alpha, A) := \max_{j \in A} t_j(\alpha(j)) \), and \( h_g(A^C) := \max_{j \in A^C} p_j \) be the maximum processing times on CPU and GPU respectively, given an assignment \( \alpha : A \rightarrow [m_c] \) and a subset \( A \subseteq [n] \). Then we can rewrite \( \omega \) as follows (see the Appendix for the complete derivation):

\[
\omega = \min_{\tau \in H} \max_{A \subseteq [n]} \left\{ \tau, \min_{A \subseteq [n]} \max_{h_g(A^C) \leq \tau} \left\{ \sum_{j \in A} \frac{w_j(\gamma_j(\tau))}{m_c}, \sum_{j \in A^C} \frac{p_j}{m_g} \right\} \right\}.
\]

Here, computing the term

\[
W(\tau) := \min_{A \subseteq [n]} \max_{h_g(A^C) \leq \tau} \left\{ \sum_{j \in A} \frac{w_j(\gamma_j(\tau))}{m_c}, \sum_{j \in A^C} \frac{p_j}{m_g} \right\}
\]

for some \( \tau \in H \) can be interpreted as minimizing the maximum makespan of two unrelated machines (R2||C_max), where \( \frac{w_j(\gamma_j(\tau))}{m_c} \) is the processing time for task \( j \) on the first machine, and \( \frac{p_j}{m_g} \) for the second machine, with the additional constraint that tasks \( j \in [n] \) with \( p_j > \tau \) must be assigned to the first machine, which can be modeled by setting their processing time for the second machine to \(+\infty\).

Now note that the identity mapping \( \tau \mapsto \tau \) is increasing, while \( \tau \mapsto W(\tau) \) is non-increasing. So one can find \( \min_{\tau \in H} \max\{\tau, W(\tau)\} \) by binary searching over the sorted processing times in \( H \), as described by Ludwig and Tiwari in [16].

The problem here is that we can not compute \( W(\tau) \) in polynomial time, since makespan minimization for two identical machines is already \( \text{NP}\)-complete [15]. But the greedy algorithm, that assigns each task to the machine on which it takes the least time, is a 2-approximate algorithm for \( R2||C_{max} \) [15].

When using a 2-approximation \( \bar{W}(\tau) \) with \( W(\tau) \leq \bar{W}(\tau) \leq 2W(\tau) \), the binary search yields a 2-approximation \( \bar{\omega} \) of \( \omega \) with \( \omega \leq \bar{\omega} \leq 2\omega \). Thus

\[
\bar{\omega}/2 \leq \omega \leq d^* \leq 2\omega \leq 2\bar{\omega}.
\]

So \( \bar{\omega} \) satisfies our criteria.
Finally, we analyze the complexity of computing $\tilde{\omega}$. Sorting $H$ is in $O((m_c n + m_g) \log (m_c n + m_g))$. The binary search takes $O(\log (m_c n + m_g))$ steps, each step costing $O(n)$ to solve the unrelated machine scheduling and $O(n \log n_c)$ to find $\gamma_j(\tau)$ for each task $j$ via binary search. Therefore the overall runtime is dominated by sorting $H$. ▶

Now that we can bound the optimal makespan for CPU/GPU scheduling, the next step is to derive a $3/2$-dual approximate algorithm. We closely follow the approach from [17].

Let $d$ be a makespan guess of some CPU/GPU scheduling instance. If a schedule of length at most $d$ exists, then a $3/2$-dual approximate algorithm has to find a schedule of length at most $3d/2$, but else it can reject the guess. We can exploit the assumption, that a schedule of at most $d$ exists, to reduce the search space by finding properties, that such schedules must satisfy and restrict the search to those. We list some basic such properties:

- **Remark 4 (structure of schedules with bounded length).** Every CPU/GPU schedule of length at most $d \in \mathbb{Q}_{>0}$ satisfies:
  1. The processing time of each task in that schedule is at most $d$, the total work area on the CPU is at most $m_c d$, and the total work area in the GPU is at most $m_g d$.
  2. If two tasks are scheduled to the CPU on the same machine, then at least one of them has a processing time at most $d/2$. This implies, that tasks on the CPU with more processing time than $d/2$ use at most $m_c$ machines in total.
  3. For each machine on the GPU: First, if a task with processing time more than $2d/3$ is scheduled to it, then other tasks scheduled to it have processing time at most $d/3$. Second, at most two tasks with processing time more than $d/3$ can be scheduled on it.

Derived from Remark 4, the basic idea from [17] for CPU scheduling is to first partition the tasks into two sets, $T_1$ for tasks with processing time more than $d/2$, and $T_2$ for those with time at most $d/2$, such that the properties from Remark 4 are satisfied, and then compute a schedule from that partition. Note that the processing time of a task depends on the number of machines they are assigned to. But by also noting, that if those properties are satisfied under some assignment of tasks to machines, then they stay satisfied when assigning each task $j$ from $T_1$ to $\gamma_j(d)$ machines, and from $T_2$ to $\gamma_j(d/2)$. In particular, they showed:

- **Lemma 5 (BuildFeasible [17]).** Let $I = (n,m, (t_j)_{j \in [n]})$ be an instance of CPU scheduling and $d \in \mathbb{Q}_{>0}$ a makespan guess. Given a partitioning of tasks $T_1 \sqcup T_2 = [n]$ with
  \[
  \sum_{j \in T_1} \gamma_j(d) \leq m \quad \text{and} \quad \sum_{j \in T_1} w_j(\gamma_j(d)) + \sum_{j \in T_2} w_j(\gamma_j(d/2)) \leq md
  \]
  one can compute a schedule of length at most $3d/2$ in time $O(mn)$.

We can extend their idea to CPU/GPU scheduling by adding a third partition $T_g$ for tasks that are to be scheduled on the GPU. Now it is left to show how to compute such a partitioning, and how to compute a schedule of length at most $3d/2$ on the GPU from $T_g$.

### 2.2 The $3/2$-dual approximate algorithm

Next, we describe how to compute a partitioning into sets $T_1$, $T_2$, $T_g$ as described previously, that satisfies the properties from Remark 4. As preparation we want to express also Property 3 as a simple inequality constraint as we did Property 2. For that, we define a size function that classifies tasks as being either small, medium, or big on the GPU.

- **Definition 6 (size function $\varphi$).** Let there be an instance of CPU/GPU scheduling and a makespan guess $d$. Then we define the size function $\varphi : [n] \to \{0,1,2, +\infty\}$ by $\varphi(j) := \max\{k \in \{0,1,2\} : kd/3 < p_j\}$ if $p_j \leq d$, and $\varphi(j) := +\infty$ if $p_j > d$. 


Then Property 3 and \( p_j \leq d \) for every \( j \in [n] \) implies, that the total size of tasks on the GPU is at most \( 2m_g \). The remaining properties of Remark 4 can be expressed by the following constraints:

\[ \sum_{j \in T_1} w_j(\gamma_j(d)) + \sum_{j \in T_2} w_j(\gamma_j(d/2)) \leq m_c d \]

\[ \sum_{j \in T_1} \gamma_j(d) \leq m_c \]

\[ \sum_{j \in T_2} \varphi(j) \leq 2m_g \]

\[ \sum_{j \in T_3} p_j \leq m_g d \]

Now, a core insight is that partitioning the tasks under these constraints can be modeled as a multidimensional multiple-choice knapsack problem (MMCKP), as described in [14]. Here we interpret the first constraint as the cost that has to be minimized, and the other three constraints from Definition 7 are represented as weights in the MMCKP, where the right-hand sides (RHSs) of Constraints 2-4 can be seen as a capacity vector \((a,b,c)\). For now, we ignore that for a knapsack formulation weights must be integer, and thus the instance processing times would need to be scaled first. While the MMCKP is \(\text{NP}\)-hard, there is a pseudopolynomial algorithm that uses dynamic programming and runs in time \(O(n \cdot abc)\) [14], similar to the standard 0-1-knapsack problem. Here, the problem is, that the two RHSs \(m_c d\) and \(m_g d\) are in general not polynomial in \(n, m_c, m_g\), and thus neither would be the complexity of solving the knapsack, since at least one of them would turn up in the capacity vector. We manage this by choosing Constraint 1 as the minimization target and employing a scaling and rounding scheme for Constraint 4, that bounds its RHS but introduces a rounding error. Next, we define the scaled and rounded constraint and discuss its choice afterward.

\[ \sum_{j \in T_g} \tilde{p}_j \leq \mu \cdot m_g d = n/\delta. \]

We need these definitions to satisfy some requirements that are critical for us. First, the \( \tilde{p}_j \) are integers, which is necessary for the dynamic program that solves the MMCKP. Second, for a constant \( \delta \in \mathbb{Q}_{>,0} \) the RHS is a small polynomial in \( n, m_c, m_g \) such that we can solve the MMCKP efficiently. These two are obviously satisfied. But third, since we want to compute a GPU schedule of length at most \( 3d/2 \) from a partitioning that satisfies the knapsack constraints, we also need the constraint to imply a good bound on the total work area on the GPU. For example, if the total work area on the GPU of the tasks in \( T_g \) is more than \( 2dm_g \), then deriving a GPU schedule of length at most \( 3d/2 \) would be impossible. The next lemma shows the bound that Constraint 5 implies for the total work on the GPU, which puts it in relation to Constraint 4.

\[ \sum_{j \in T_g} p_j > m_g d. \]
Scheduling on CPU/GPU

**Proof.** The definition of $\tilde{p}_j$ yields $\tilde{p}_j/\mu \leq p_j \leq (\tilde{p}_j + 1)/\mu$ for all $j \in [n]$. If Constraint 5 holds:

$$\sum_{j \in T_g} p_j \leq \sum_{j \in T_g} \left( \frac{\tilde{p}_j}{\mu} + \frac{1}{\mu} \right) \leq \frac{1}{\mu} \sum_{j \in T_g} \tilde{p}_j + \frac{n}{\mu} = m_g \cdot (1 + \delta) d,$$

and if Constraint 5 does not hold: $\sum_{j \in T_g} p_j \geq \frac{1}{\mu} \sum_{j \in T_g} \tilde{p}_j > \frac{1}{\mu} \cdot \frac{n}{d} = m_g d$. □

The preceding Lemma 9 shows that the scaling and rounding scheme implies a relaxation of Constraint 4 by the factor $(1 + \delta)$. Therefore reducing the error coefficient $\delta$ improves the bound on the total work by reducing the rounding error, but increases the time complexity of solving the MMCKP by increasing the RHS. The maximal possible choice of $\delta$ depends on the algorithm used to build a GPU schedule from the partition $T_g$. Next, we describe this algorithm and bound the makespan of the resulting schedule under the assumption that Constraint 3 and a relaxed version of Constraint 4 hold.

**Lemma 10 (Build feasible schedule for the GPU).** Let there be a CPU/GPU scheduling instance, a makespan guess $d \in \mathbb{Q}_{>0}$, $c \geq 0$, and $T_g \subseteq [n]$. If Constraint 3 holds, and $\sum_{j \in T_g} p_j \leq m_g (1 + c) d$, then we can construct a GPU schedule of length at most $(4/3 + c) d$ in time $\mathcal{O}(n \log n)$.

**Proof.** We can build a schedule with makespan at most $(4/3 + c) d$ using LPT-scheduling:

1. Sort the tasks $j \in T_g$ by descending processing times $p_j$.
2. In that order, assign each task to the machine with the smallest workload.

This algorithm runs in time $\mathcal{O}(n \log n)$ since the runtime is dominated by sorting the tasks. Now, we bound the makespan of the resulting schedule by $(4/3 + c) d$.

With LPT, the tasks $j \in T_g$ of size $\varphi(j) = 2$ get assigned first. Since by Constraint 3 there can be at most $m_g$ such tasks in $T_g$, every machine is assigned at most one task and therefore has a workload of at most $d$.

Next, the tasks $j \in T_g$ of size $\varphi(j) = 1$ get assigned. Suppose task $j \in T_g$ is the first such task that is assigned to a machine with a load of more than $(2/3) d$. Then, by construction, all machines have a load of more than $(2/3) d$. If a machine has a load of more than $(2/3) d$ while only tasks of size 2 or 1 are assigned to it, then at least one task with size 2 is assigned to it, or at least two tasks with size 1. So the total size on all machines is already at least $2m_g$, excluding task $j$. Including task $j$ yields a contradiction to Constraint 3.

Thus, tasks with size 1 get assigned only to machines with a load of at most $(2/3) d$, and after assigning such a task to a machine, its load is at most $(4/3) d$.

Tasks $j \in T_g$ of size $\varphi(j) = 0$ get assigned last. When some task $j$ with size 0 gets assigned to some machine $k \in [m_g]$, such that its workload changes from less than $(1 + c) d$ to more than $(1 + c) d$, then two properties hold. First, its workload is still at most $(4/3 + c) d$. Second, no other task will get assigned to machine $k$, before all machines have a workload of more than $(1 + c) d$ because of the second step of the algorithm. If at any point in time all machines had a workload of more than $(1 + c) d$, though, then that state would contradict our assumption that $\sum_{j \in T_g} p_j \leq m_g (1 + c) d$. Thus, no task with size 0 gets assigned to a machine with a workload of more than $(1 + c) d$.

In conclusion, no step of assigning a task to a machine will increase the workload of a machine to more than $(4/3 + c) d$, and thus the algorithm constructs a schedule with makespan at most $(4/3 + c) d$. □
Now we can see that $\delta = 1/6$ is the maximal constant such that Lemma 10 guarantees a schedule of length at most $3d/2$, when given a partition $\mathcal{T}_g$ that satisfies Constraints 3 and 5.

To summarise we now define the MMCKP for CPU/GPU scheduling.

**Definition 11 (MMCKP for CPU/GPU scheduling).** Let there be a CPU/GPU scheduling instance, $d \in \mathbb{Q}_{>0}$ a makespan guess and $\delta := 1/6$. The goal of the Multidimensional Multiple Choice Knapsack Problem for CPU/GPU scheduling is to find a partitioning $\mathcal{T}_1 \cup \mathcal{T}_2 \cup \mathcal{T}_g = [n]$ that minimises the function

$$W(\mathcal{T}_1, \mathcal{T}_2) := \sum_{j \in \mathcal{T}_1} w_j(\gamma_j(d)) + \sum_{j \in \mathcal{T}_2} w_j(\gamma_j(d/2)).$$

**Remark 12.** As discussed previously, a solution of the MMCKP for CPU/GPU scheduling can be computed in time $O(nm_c2m_gn/\delta) = O(n^2m_cm_g)$ by dynamic programming [14].

Now we can describe a 3/2-dual approximate algorithm for CPU/GPU scheduling, see Algorithm 1. We begin by constructing a partition $\mathcal{T}_1 \cup \mathcal{T}_2 \cup \mathcal{T}_g = [n]$ using the multidimensional multiple choice knapsack algorithm in Remark 12. If we find a valid partition of the tasks with $W(\mathcal{T}_1, \mathcal{T}_2) \leq m_cd$, we use Lemma 5 to construct the schedule for the CPU. Next, we use Lemma 10 to build a schedule for the GPU. Finally, we combine both schedules, resulting in a CPU/GPU schedule. On the other hand, if there is no valid partition found, we can reject the makespan guess $d$ and return $\emptyset$, as we know that no feasible schedule exists for it.

**Algorithm 1** Description of the 3/2-dual approximation algorithm.

```
Require: CPU/GPU scheduling instance $I$
1: Construct $\mathcal{T}_1 \cup \mathcal{T}_2 \cup \mathcal{T}_g = [n]$ using MMCKP $\triangleright$ Remark 12
2: if $W(\mathcal{T}_1, \mathcal{T}_2) \leq m_cd$ then
3: \quad Build schedule $\sigma_c$ for CPU from $\mathcal{T}_1, \mathcal{T}_2$ $\triangleright$ Lemma 5
4: \quad Build schedule $\sigma_g$ for GPU from $\mathcal{T}_g$ $\triangleright$ Lemma 10
5: \quad Return schedule $(\sigma_c, \sigma_g)$
6: else Return $\emptyset$
7: end if
```

We continue by showing the correctness of Algorithm 1.

**Lemma 13.** Let there be a CPU/GPU scheduling instance and $d \in \mathbb{Q}_{>0}$ a makespan guess and $\delta = 1/6$. If a schedule of length at most $d$ exists, then the 3/2-dual approximate algorithm returns a schedule of length at most $3d/2$.

**Proof.** Consider a schedule of length at most $d$. The schedule implies a partitioning of tasks into those processed on the CPU with processing time more than $d/2$ (and at most $d$), those on the CPU with time at most $d/2$, and those processed on the GPU, called $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_g$ respectively. The schedule also implies a mapping $\pi : \mathcal{T}_1 \cup \mathcal{T}_2 \to \mathbb{N}$ of tasks to the number of machines on the CPU they are processed on. Then, by Remark 4 the partitioning satisfies:

- $\sum_{j \in \mathcal{T}_1 \cup \mathcal{T}_2} w_j(\pi(j)) \leq m_cd$,
- $\sum_{j \in \mathcal{T}_1} \pi(j) \leq m_c$,

as well as Constraints 3 and 4. By definition we have $\gamma_j(d) \leq \pi(j)$ for all $j \in \mathcal{T}_1$ and $\gamma_j(d/2) \leq \pi(j)$ for all $j \in \mathcal{T}_2$. Then, we get from the first point and the monotony of $w_j$, that Constraint 1 holds, and from the second point that Constraint 2, too. Finally, Constraint 5 holds, since it would otherwise violate Constraint 4 with Lemma 9.
In conclusion, the constructed partitioning is a feasible solution to the MMCKP for CPU/GPU scheduling with \( W(T_1, T_2) \leq m_c \), therefore the set of solutions is non-empty and the MMCKP algorithm delivers a feasible solution. Lemma 5 and 10 guarantee that a CPU/GPU schedule of length at most \( 3d/2 \) is returned.

2.3 Optimizing the Algorithm

If we look closely, the MMCKP for CPU/GPU scheduling processes some unnecessary or redundant information:

1. For Constraint 3 the tasks \( j \in [n] \) with size \( \varphi(j) = 0 \) are irrelevant.
2. In Definition 8 we chose \( \mu \) such that \( n \mu = \delta m_g d \), because in the proof of Lemma 9 we can bound the number of tasks in \( T_g \) only by \( n \). But the number of tasks \( j \in [n] \) with size \( \varphi(j) \geq 1 \) can be bound by \( 2m_g \) due to Constraint 3. Thus, when considering only those tasks we could choose a scaling factor \( \lambda \) such that \( 2m_g/\lambda = \delta m_g d \Leftrightarrow \lambda = 2/(\delta d) \).

When scaling Constraint 4 with \( \lambda \), then its RHS becomes \( 2m_g/\lambda \), which is smaller than \( n/\delta \) if \( 2m_g < n \).
3. The size function \( \varphi \) and Constraints 3 and 5 carry duplicate information about the processing time of tasks on the GPU. This suggests that we could reduce the RHS of the constraint for the total work on the GPU, if we cut out that redundant information.

The first two points suggest splitting the MMCKP for CPU/GPU scheduling into two stages, one for the tasks with size at least 1, and one for those with size 0. This justifies giving names to those classes of tasks. The third point can be treated separately.

Definition 14 (small and big tasks, \( B, S \)). Let there be a CPU/GPU scheduling instance.

Tasks \( j \in [n] \) with \( \varphi(j) \geq 1 \) we call big (on the GPU), and those with \( \varphi(j) = 0 \) we call small (on the GPU). Further we define the set of big tasks \( B := \varphi^{-1}([1, 2, +\infty]) \), and the set of small tasks \( S := \varphi^{-1}(0) \).

Split MMCKP

Definition 15 (\( \lambda \), Constraint 6). Let there be a CPU/GPU scheduling instance, a makespan guess \( d \in \mathbb{Q}_{>0} \), and an error coefficient \( \delta \in \mathbb{Q}_{>0} \). We define the scaling factor \( \lambda := 2 \delta d \), and the constraints

6. \( \sum_{j \in T_g} \lambda p_j \leq \lambda m_g d = 2m_g/\delta \).

We want to split the knapsack into two stages, the first processing only big tasks \( j \in B \) and the second only processing small tasks \( j \in S \). But we also need to compose them in some way, to ensure global optimality of the resulting partitioning, and not just locally for each set of big or small tasks. The dynamic programming algorithm for the MMCKP internally computes a table, where each entry encodes a solution for a smaller subproblem [14]. So we use the whole table that the big task knapsack computes as an input for the small task knapsack. Then the multiple-choice knapsack for the small tasks first has to choose a partitioning of the big tasks, among those encoded in that table.

Now we define the knapsack for big tasks. We will see later in Remark 19 how to choose the error coefficient \( \delta \).

Definition 16 (Big Tasks MMCKP for CPU/GPU scheduling). Let there be a CPU/GPU scheduling instance, a makespan guess \( d \in \mathbb{Q}_{>0} \), and an error coefficient \( \delta \in \mathbb{Q}_{>0} \). Let \( B = \{b_1, \ldots, b_K\} \) be an enumeration.
One goal of the Big Tasks Multidimensional Multiple Choice Knapsack Problem for CPU/GPU scheduling is to compute a table $W^*_B$, where for all $(j, a, b, c) \in [K] \times [m_c]_0 \times [2m_g]_0 \times [2m_g/\delta]_0$ the entry $W^*_B(j, a, b, c)$ minimises the target function $W(T_1, T_2) := \sum_{j \in T_1} w_j(\gamma_j(d)) + \sum_{j \in T_2} w_j(\gamma_j(d/2))$

over all partitions $T_1 \sqcup T_2 \sqcup T_g = \{b_1, \ldots, b_\lambda\}$ such that the left-hand-sides of Constraints 2, 3, and 6 are equal to $a$, $b$, and $c$ respectively.

Another goal is to compute a minimizing partitioning given such a tuple $(j, a, b, c)$.

$W^*_B(K, a, b, c)$ encodes a partitioning of $B$ that satisfies above properties. The knapsack for small tasks first chooses one of those entries. The total work on the GPU, encoded in term $c$ by the big tasks knapsack is scaled by factor $\lambda$, but the small tasks knapsack uses the scaling factor $\mu$. Thus we need to re-scale that information to use it in the constraint for the small tasks knapsack:

Definition 17 (Small Tasks MMCKP for CPU/GPU scheduling). Let there be a CPU/GPU scheduling instance, a makespan guess $d \in \mathbb{Q}_{>0}$, and an error coefficient $\delta \in \mathbb{Q}_{>0}$. Let the table $W^*_B$ be a solution to the big tasks MMCKP for CPU/GPU scheduling. The goal of the Small Tasks Multidimensional Multiple Choice Knapsack Problem for CPU/GPU scheduling is to choose a triple $(a, b, c) \in [m_c]_0 \times [2m_g]_0 \times [2m_g/\delta]_0$ and a partitioning $T_1 \sqcup T_2 \sqcup T_g = S$ that minimizes $W_S((a, b, c), T_1, T_2) := W^*_B(K, a, b, c) + \sum_{j \in T_1} w_j(\gamma_j(d)) + \sum_{j \in T_2} w_j(\gamma_j(d/2))$

under the following modifications of Constraints 2 and 5:

- $a + \sum_{j \in T_1} \gamma_j(d) \leq m_c$, and
- $\lceil \frac{\mu}{\lambda} c \rceil + \sum_{j \in T_g} \tilde{p}_j \leq n/\delta$.

Then, in total, we compute a partitioning $T_1 \sqcup T_2 \sqcup T_g = [n]$ by first computing the table $W^*_B$ from the big tasks MMCKP, then with that table we compute a triple $(a, b, c)$ and a partitioning $T'_1 \sqcup T'_2 \sqcup T'_g = S$ from the small tasks MMCKP, and then with the tuple $(K, a, b, c)$, where $K = |B|$, we compute a partitioning $T''_1 \sqcup T''_2 \sqcup T''_g = B$ from the big tasks MMCKP. Finally we set $T_1 := T'_1 \sqcup T'_1$, $T_2 := T'_2 \sqcup T'_2$, and $T_g := T'_1 \sqcup T'_g$.

We still need to show that the scaled and rounded constraint for the total work on the GPU in the small tasks MMCKP implies a good bound on the actual work area, as we did in the previous section.

Lemma 18. Let there be a CPU/GPU scheduling instance, a makespan guess $d \in \mathbb{Q}_{>0}$, and an error coefficient $\delta \in \mathbb{Q}_{>0}$. Let $T_s \subseteq S$ such that Constraints 3 and 6 hold. Let $T_g \subseteq S$ such that $\left\lfloor \frac{\delta}{n} \sum_{j \in T_s} \bar{p}_j \right\rfloor + \sum_{j \in T_g} \tilde{p}_j \leq n/\delta$ holds. Then $\sum_{j \in T_s \cup T_g} p_j \leq m_g (1 + 2\delta + \delta/n)d$.

Proof. (See the Appendix.)

Remark 19. Lemma 18 shows that if we set the error coefficient to $\delta := 1/[12 + \frac{\delta}{n}]$, then Lemma 10 builds a schedule of length at most $3d/2$ from the set of tasks $T_g := T_b \cup T_s$.

We skip the correctness proof for the dual approximation that uses the split MMCKP, as it works with arguments analogous to those in Lemma 13.
Cutting redundant information

In this section, we show how to decrease the RHS of the constraint that represents the total work area on the GPU in the big tasks MMCKP, which improves the time complexity of solving it. First, we describe how to modify the knapsack to achieve this, and then analyze the time savings.

Definition 20 (Modified Big and Small Tasks MMCKP for CPU/GPU scheduling, β). The modified big tasks MMCKP for CPU/GPU scheduling is the unmodified one, except with the following modifications. First, define \( \beta := \lfloor \lambda d/3 \rfloor \), which is the scaled and rounded work area represented by one size step of \( \varphi \).

- The table is defined for all tuples \((j, a, b, c)\) such that \((j, a, b) \in [K] \times [m_c]_0 \times [2m_g]_0\) and \(c \in [2m_g/\delta - b\beta]_0\).
- Instead of Constraint 6 we use the constraint \(\sum_{j \in T_g} \lfloor \lambda p_j \rfloor - \varphi(j)\beta = c\)

The modified small tasks MMCKP for CPU/GPU scheduling is modified such that

- We choose a triple \((a, b, c)\) with \((a, b) \in [m_c]_0 \times [2m_g]_0\) and \(c \in [2m_g/\delta - b\beta]_0\) instead.
- We use the constraint \(\frac{\lambda}{\beta} (c + \beta b) + \sum_{j \in T_g} \tilde{p}_j \leq n/\delta\) as the modification of Constraint 5.

This modification decreases the size of the table \(W^*_g\) to almost \(2/3\) of its original size. While this does not improve the time complexity in \(O\)-notation, this is a significant speedup for implementations of the algorithm, since the main computational cost of solving the knapsack by dynamic programming lies in computing every entry of the table. In the next lemma, we prove the stated speedup factor.

Lemma 21. Let there be a CPU/GPU scheduling instance. Let \(W^*\) the table from the big tasks MMCKP, and \(X^*\) from the modified one. Let the mapping \(|\cdot|\) count their number of entries. Then \(\frac{|X^*|}{|W^*|} \leq \frac{2}{3}(1 + \delta)\).

Proof. (For a more detailed calculation see the Appendix.) Let \(K := [B]\). We have \(|W^*| = K(m_c+1)(2m_g+b)(2m_g/\delta+1)\). Further, we have \(|X^*| = K(m_c+1)(2m_g+1)(2m_g/\delta+1-\beta m_g)\). Now, with \(\beta = \lfloor 2/(3\delta) \rfloor \geq 2/(3\delta) - 1\) we get

\[
(2m_g/\delta + 1 - \beta m_g) \leq (2/3)(2m_g/\delta + 1) + (1/3) + m_g.
\]

Then one can show, that \(\frac{|X^*|}{|W^*|} \leq \frac{2}{3} + \frac{1}{2\epsilon} \cdot \frac{1}{2\epsilon} = \frac{2}{3}(1 + \delta)\).

3 Complexity

We showed that we can compute a bound \(\omega\) on the optimal makespan \(d^*\) such that such that \(\omega \leq d^* \leq 4\omega\) in time \(O((m_c n + m_g) \log(m_c n + m_g))\).

The complexity of the 3/2-dual approximation for both, the unmodified and the modified version, is dominated by the complexity of solving their knapsack problems. Solving the unmodified MMCKP is in \(O(n^2 m_c m_g)\), see Remark 12. Solving the modified MMCKP for big tasks is in \(O(n m_c m_g^2)\), for small tasks in \(O(n^2 m_c)\), so solving both is in \(O(n m_c \cdot \max(n, m_g^2))\).

Therefore, with Lemma 2, computing a \((3/2 + \epsilon)\)-approximation for the CPU/GPU scheduling is in total

\[
O((m_c n + m_g) \log(m_c n + m_g)) + \log(1/\epsilon) n^2 m_c m_g \subseteq O(\log(n m_c m_g/\epsilon) \cdot n^2 m_c m_g)
\]
for the unmodified version, and \(O(\log(n m_c m_g/\epsilon) \cdot n m_c \cdot \max(n, m_g^2))\) for the modified version.
4 Implementation

To complement our theoretical improvements on the current state of the art, we implemented both the naive and improved version of our algorithm. In the following, we describe the implementation and compare these implementations not only to each other but also to the current state of the art. The implementation is in Java and publicly available on GitHub (https://github.com/Felioh/CPU-GPU-Scheduling) as well as the test results.

4.1 Test Environment

The experiments were run on the High-Performance-Cluster of the University of Kiel, more specifically on a node with 2 AMD Epyc 7313 CPUs (32 Cores) and 512GB main memory. We disabled the Java Garbage Collector, as the default is non-deterministic. It is important to note that obtaining meaningful results can be challenging due to the various parameters involved, including the number of CPU machines, the number of GPU machines, the number of tasks, and the processing time required for each task on both the CPU and GPU machines. We primarily used randomly generated instances to ensure the robustness of the results across a large number of instances.

4.2 Results

We conducted two sets of tests: one to demonstrate the dependence on machine numbers and the other to show the dependence on task numbers. For both test runs we set $\varepsilon = 0.1$ and the processing time of all tasks to be randomly distributed in the interval [20, 100]. The processing time of GPU tasks on one machine is generated in the same way. The processing time for more machines is generated by respecting the given constraints. Additionally, the number of GPU machines was always set to 20% of the number of CPU machines as we believe this relationship could significantly impact the running time. All test results are available in the corresponding GitHub Repository.

To test the effect of the number of machines on our instances, we created instances with a minimum of 10 and a maximum of 100 CPU machines. To maintain consistency, we restricted the number of tasks to 50. The results of these tests are shown in Figure 2.

To examine how the number of tasks affects the total runtime of our algorithm, we generated instances with 50 CPU machines and the number of tasks varying from 10 to 100. The results of these tests are shown in Figure 3.

The results indicate a greater improvement than expected, likely due to inefficiencies in memory accesses caused by the naive implementation’s increased memory requirements.

Table 1 shows the comparison of our implementation and the implementation by Bleuse et al. [1]. To achieve these results we tested both implementations with the same input instances. We can observe that our implementation is significantly slower for small instances. As the instances get bigger our implementation catches up with the trade-off of using more memory. The instances used for these tests and the full results are available on GitHub.

5 Conclusion

We presented an efficient algorithm to approximate the maximum makespan of the GPU/CPU scheduling problem with a factor of $(3/2 + \varepsilon)$, using the dual approximation framework. We improve the asymptotic running time of the dual approximation from a theorized [1] time of $O(n^2m_1m_2^2)$ to $O(n^2m_1m_2)$. Impressively, we manage to reduce the dependency on the number of CPU-machines by $m_2^3$, while also reducing the dependence on GPU-machines by
Our implementation utilizes a knapsack formulation to assign tasks. Solving this knapsack dominates the running time. To improve it, we observe that large parts of information are superfluous. This does not strictly improve the asymptotic running time, resulting in a time of $O(nm_c \cdot \max(n, mg)^2)$ for the dual approximation. However, we show that these practical improvements have a large impact on the execution time of the algorithm.

In future work, the runtime could be further improved by applying compression techniques from [6], using a more sophisticated algorithm to solve the MMCKP, or using two different algorithms depending on the sizes of $m_c, mg$.

Further, we compare our implementation to another existing state of the art algorithm [1], that makes use of commercial solvers like CPLEX, Gurobi, or GLPK and show that our implementation is faster for big instances while keeping the same makespan guarantee.

References


### A Proofs

**Proof of Lemma 3.** Let $t_j : [m_c] \rightarrow \mathbb{Q}_{>0}$ be the processing time function, $w_j$ the work area function on the CPU, and $p_j$ the processing time on the GPU for each task $j \in [n]$. Now we define the mapping $\omega : [m_c]^{[n]} \times \mathcal{P}([n]) \rightarrow \mathbb{Q}_{>0}$ by

$$\omega(\alpha, A) := \max \left\{ \frac{1}{m_c} \sum_{j \in A} w_j(\alpha(j)), \max_{j \in A} \{t_j(\alpha(j))\}, \frac{1}{m_g} \sum_{j \in A^C} p_j, \max_{j \in A^C} \{p_j\} \right\}.$$ 

There
- $A \subseteq [n]$ represents the tasks scheduled on the CPU and its complement $A^C = [n]/A$ on the GPU,
- the mapping $\alpha : A \rightarrow [m_c]$ assigns to each task a number of CPU machines,
- $\frac{1}{m_c} \sum_{j \in A} w_j(\alpha(j))$ is the average work per machine on the CPU,
- $\max_{j \in A} \{t_j(\alpha(j))\}$ is the maximum processing time of any task on the CPU,
- $\frac{1}{m_g} \sum_{j \in A^C} p_j$ is the average work per machine on the GPU, and
- $\max_{j \in A^C} \{p_j\}$ is the maximum processing time of any task on the GPU.

Given an assignment $\alpha$ and a partition $A \cup A^C = [n]$, the 2-approximate algorithm by Garey and Graham yields a CPU/GPU schedule with a makespan that does not exceed $2\omega(\alpha, A)$ [4]. This suggests the following strategy to find the desired lower bound: Find an assignment and partition such that $\omega(\alpha, A) \leq d^* \leq 2\omega(\alpha, A)$.

Now, $\omega$ defined by

$$\omega := \min_{A \subseteq [n]} \min_{\alpha : [n] \rightarrow [m_c]} \omega(\alpha, A)$$ 

is a trivial lower bound on the minimum makespan for CPU/GPU scheduling, and, by Garey and Graham’s algorithm, satisfies $\omega \leq d^* \leq 2\omega$. Next, we will derive an efficient way to compute $\omega$. 


Let $H := \{p_j : j \in [n]\} \cup \{t_j(p) : j \in [n], p \in [m_c]\}$ be the set of processing times of tasks on the CPU/GPU. Further, let $h_\alpha(\alpha, A) := \max_{j \in A} t_j(\alpha(j))$, and $h_\alpha(A^C) := \max_{j \in A^C} p_j$ be the maximum processing times on CPU and GPU respectively, given an assignment $\alpha : A \rightarrow [m_c]$ and a subset $A \subseteq [n]$. Then we can rewrite $\omega$ as follows:

$$ \omega = \min_{A \subseteq [n]} \min_{\alpha : A \rightarrow [m_c]} \omega(\alpha, A) $$

$$ = \min_{A \subseteq [n]} \min_{\alpha : A \rightarrow [m_c]} \max \left\{ \frac{1}{m_c} \sum_{j \in A} w_j(\alpha(j)), \max\{t_j(\alpha(j))\}, \frac{1}{m_g} \sum_{j \in A^C} p_j, \max\{p_j\} \right\} $$

$$ = \min_{\tau \in H} \min_{A \subseteq [n]} \max_{\alpha : A \rightarrow [m_c]} \left( \frac{1}{m_c} \sum_{j \in A} w_j(\alpha(j)), \max\{t_j(\alpha(j))\}, \frac{1}{m_g} \sum_{j \in A^C} p_j \right) $$

$$ = \min_{\tau \in H} \min_{A \subseteq [n]} \max_{\alpha : A \rightarrow [m_c]} \left( \frac{1}{m_c} \sum_{j \in A} w_j(\alpha(j)), \max\{t_j(\alpha(j))\}, \frac{1}{m_g} \sum_{j \in A^C} p_j \right) $$

$$ = \min_{\tau \in H} \max_{\alpha : A \rightarrow [m_c]} \left( \frac{1}{m_c} \sum_{j \in A} w_j(\alpha(j)), \max\{t_j(\alpha(j))\}, \frac{1}{m_g} \sum_{j \in A^C} p_j \right) $$

Here, computing the term

$$ W(\tau) := \min_{\alpha : A \rightarrow [m_c]} \max_{\tau \in H} \left\{ \frac{w_j(\gamma_j(\tau))}{m_c}, \frac{1}{m_g} \sum_{j \in A^C} p_j \right\} $$

for some $\tau \in H$ can be interpreted as minimizing the maximum makespan of two unrelated machines $(R2|C_{\text{max}})$, where $w_j(\gamma_j(\tau))$ is the processing time for task $j$ on the first machine, and $\frac{1}{m_g}$ for the second machine, with the additional constraint that tasks $j \in [n]$ with $p_j > \tau$ must be assigned to the first machine, which can be modeled by setting their processing time for the second machine to $+\infty$.

Now note that the identity mapping $\tau \mapsto \tau$ is increasing, while $\tau \mapsto W(\tau)$ is non-increasing. So one can find $\min_{\tau \in H} \max\{\tau, W(\tau)\}$ by binary searching over the sorted processing times in $H$, as described by Ludwig and Tiwari in [16].

The problem here is that we cannot compute $W(\tau)$ in polynomial time, since makespan minimization for two identical machines is already $\text{NP}$-complete [15]. But the greedy algorithm, that assigns each task to the machine on which it takes the least time, is a 2-approximate algorithm for $R2|C_{\text{max}}$ [15].

When using a 2-approximation $\overline{W}(\tau)$ with $W(\tau) \leq \overline{W}(\tau) \leq 2W(\tau)$, the binary search yields a 2-approximation $\overline{\omega}$ of $\omega$ with $\omega \leq \overline{\omega} \leq 2\omega$. Thus

$$ \frac{\overline{\omega}}{2} \leq \omega \leq \frac{\overline{\omega}}{2} $$

So $\overline{\omega}$ satisfies our criteria.

Finally, we analyze the complexity of computing $\overline{\omega}$. Sorting $H$ is in $O((m_c n + m_g) \log(m_c n + m_g))$. The binary search takes $O(\log(m_c n + m_g))$ steps, each step costing $O(n)$ to solve the unrelated machine scheduling and $O(n \log m_c)$ to find $\gamma_j(\tau)$ for each task $j$ via binary search. Therefore the overall runtime is dominated by sorting $H$. \qed
Proof of Lemma 18. The definition of $\tilde{p}_j$ yields $\tilde{p}_j/\mu \leq p_j \leq (\tilde{p}_j + 1)/\mu$ for all $j \in T_s$, and analogously we also have $|\lambda p_j|/\lambda \leq p_j \leq (|\lambda p_j| + 1)/\lambda$ for all $j \in T_b$. Note, that from Constraint 3 follows, that $|T_b| \leq 2m_g$. Then:

$$\sum_{j \in T_b \cup T_s} p_j = \sum_{j \in T_b} p_j + \sum_{j \in T_s} p_j \leq \left( \frac{1}{\lambda} \sum_{j \in T_b} |\lambda p_j| \right) + \frac{2m_g}{\lambda} + \left( \frac{1}{\mu} \sum_{j \in T_s} \tilde{p}_j \right) + \frac{n}{\mu} \leq \frac{1}{\mu} \left( \frac{\mu}{\lambda} \sum_{j \in T_b} |\lambda p_j| + \sum_{j \in T_s} \tilde{p}_j \right) + \frac{2m_g}{\lambda} + \frac{n}{\mu} \leq \frac{1}{\mu} \left( 1 + \left\lfloor \frac{\mu}{\lambda} \sum_{j \in T_b} |\lambda p_j| \right\rfloor + \sum_{j \in T_s} \tilde{p}_j \right) + \frac{2m_g}{\lambda} + \frac{n}{\mu} \leq \frac{1}{\mu} + \frac{n}{\mu} \delta + \frac{2m_g}{\lambda} + \frac{n}{\mu} = \frac{m_g \delta d}{n} + \frac{nm_g \delta d}{n\delta} + \frac{2m_g \delta d}{2} + \frac{nm_g \delta d}{n} = m_g (1 + 2\delta + \frac{\delta}{n})d \leq 1.\] 

Proof of Lemma 21. Let $K := |B|$. We have $|W^*| = K(m_c + 1)(2m_g + 1)(2m_g/\delta + 1)$. Further, we have

$$|X^*| = K(m_c + 1) \sum_{0 \leq b \leq 2m_g} (2m_g/\delta + 1 - b\beta) = K(m_c + 1) \left( (2m_c + 1)(2m_g/\delta + 1) - \beta \sum_{1 \leq b \leq 2m_g} b \right) = K(m_c + 1)(2m_g + 1)(2m_g/\delta + 1 - \beta m_g)$$

Now, with $\beta = [2/(3\delta)] \geq 2/(3\delta) - 1$ we get

$$(2m_g/\delta + 1 - \beta m_g) \leq \frac{2}{3} (2m_g/\delta + 1) + \frac{1}{3} + m_g.$$ 

Concluding:

$$\frac{|X^*|}{|W^*|} = \frac{2m_g/\delta + 1 - \beta m_g}{2m_g/\delta + 1} \leq \frac{2}{3} + \frac{1}{3} \cdot \frac{1}{2m_g/\delta + 1} + \frac{m_g}{2m_g/\delta + 1} \leq \frac{2}{3} + \frac{4}{3} \cdot \frac{1}{2\delta} = \frac{2}{3} (1 + \delta). \leq 1.$$
B  Figures and Tables

- **Figure 2** Runtime in relation to machine numbers.

- **Figure 3** Runtime in relation to task numbers.

- **Table 1** Comparison of makespan and runtime with [1] for $100 \leq n \leq 400$ and $\varepsilon = 0.1$.

<table>
<thead>
<tr>
<th>$m_c$</th>
<th>$m_g$</th>
<th>$n$</th>
<th>our time [s]</th>
<th>time of [1] [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>20</td>
<td>100</td>
<td>19.7</td>
<td>1.0</td>
</tr>
<tr>
<td>100</td>
<td>20</td>
<td>150</td>
<td>28.8</td>
<td>2.1</td>
</tr>
<tr>
<td>100</td>
<td>20</td>
<td>200</td>
<td>41.7</td>
<td>20.9</td>
</tr>
<tr>
<td>100</td>
<td>20</td>
<td>250</td>
<td>67.1</td>
<td>18.2</td>
</tr>
<tr>
<td>100</td>
<td>20</td>
<td>300</td>
<td>95.6</td>
<td>32.4</td>
</tr>
<tr>
<td>100</td>
<td>20</td>
<td>350</td>
<td>93.6</td>
<td>183.0</td>
</tr>
<tr>
<td>100</td>
<td>20</td>
<td>400</td>
<td>127.6</td>
<td>199.8</td>
</tr>
</tbody>
</table>
Accelerating ILP Solvers for Minimum Flow Decompositions Through Search Space and Dimensionality Reductions

Andreas Grigorjew
University of Helsinki, Finland

Fernando H. C. Dias
Aalto University, Espoo, Finland

Andrea Cracco
University of Verona, Italy

Romeo Rizzi
University of Verona, Italy

Alexandru I. Tomescu
University of Helsinki, Finland

Abstract

Given a flow network, the Minimum Flow Decomposition (MFD) problem is finding the smallest possible set of weighted paths whose superposition equals the flow. It is a classical, strongly NP-hard problem that is proven to be useful in RNA transcript assembly and applications outside of Bioinformatics.

We improve an existing ILP (Integer Linear Programming) model by Dias et al. [RECOMB 2022] for DAGs by decreasing the solver’s search space using solution safety and several other optimizations. This results in a significant speedup compared to the original ILP, of up to 34× on average on the hardest instances. Moreover, we show that our optimizations apply also to MFD problem variants, resulting in speedups that go up to 219× on the hardest instances.

We also developed an ILP model of reduced dimensionality for an MFD variant in which the solution path weights are restricted to a given set. This model can find an optimal MFD solution for most instances, and overall, its accuracy significantly outperforms that of previous greedy algorithms while being up to an order of magnitude faster than our optimized ILP.

2012 ACM Subject Classification Theory of computation → Network flows; Applied computing → Bioinformatics

Keywords and phrases Flow decomposition, Integer Linear Programming, Safety, RNA-seq, RNA transcript assembly, isoform

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.14

Supplementary Material Software (Source Code, Datasets): https://github.com/algbio/optimized-fd, archived at swh:1:dir:1e7c3ad9381a18899122a662b79914eb33cece80

Funding This work was partially funded by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No. 851093, SAFEBIO), and by the Research Council of Finland (grants No. 322595, 352821, 346968, 358744).

Acknowledgements We are very grateful to Manuel Cáceres for a very helpful discussion on sets of independent safe paths.
1 Introduction

Motivation. Minimum Flow Decomposition (MFD) is the problem of finding a minimum number of weighted paths to decompose a flow on a directed graph, such that the sum of weights of the paths crossing an edge is equal to the flow on that edge. We refer to this number as the size of the MFD. Hartman et al. [9] proved that MFD is NP-hard, even on directed acyclic graphs (DAGs), and even if the flow values are only in \{1, 2, 4\}.

MFD is a crucial tool in many applications, for example, in networking, where the paths represent traffic going through a sequence of routers. Minimizing the number of these paths reduces the amount of maintenance work, among other additional benefits [9]. In RNA sequence reconstruction problems, MFD is used by e.g. [16, 22, 6, 21, 25] to decompose a splice graph, in which every node is a gene exon, and every weighted path is an RNA sequence that needs to be reconstructed. This splice graph can, for example, be constructed by aligning RNA-seq reads to a reference genome. MFD can also used to reconstruct viral quasispecies [1]. The input graphs in these applications are DAGs, and all previous work on MFD that we cite below is also on DAGs. As such, in this paper, all graphs are DAGs.

Moreover, MFD variants are of practical interest. For example, one practical strategy is to incorporate longer reads as subpaths potentially spanning more than two nodes [16, 19, 26, 8], which must appear in some solution path (i.e. reconstructed transcript or viral genome). Alternatively, the edge weights may not form an exact flow, due to sequencing errors and read mapping artifacts. An MFD variant adapted to this case is to consider intervals of edge weights instead, called inexact flows in [18, 25].

Related work. Given the complexity of the problem, it is common to use heuristic methods [20], for example, by using a greedy approach, iteratively removing the path with the largest currently available flow until the flow is fully decomposed. Cáceres et al. [17] showed that for DAGs this approach performs well, with an approximation ratio of \(O(\log \text{val}(f))\) (where \(\text{val}(f)\) is the total flow of the graph), only if the width of the graph (the minimum number of paths to cover all edges) does not increase in the process; otherwise it gives an exponentially worse result than the optimal with an approximation ratio of \(\Omega(m/\log m)\).

Mumey et al. [14] gave an approximation algorithm on DAGs with ratio \(\lambda \log \|f\| \log \|f\|\), where \(\lambda\) is the longest source-to-sink path length and \(\|f\|\) is the maximum-norm on the flow \(f\) (i.e., the largest flow value of the graph). Polynomial-time approximation algorithms with sublinear ratio in the size of the graph are not known, neither whether the problem is in APX, i.e. admits a constant-factor approximation algorithm.

Kloster et al. [11] proposed the first algorithm solving MFD optimally, which is linear-time fixed-parameter tractable (FPT), parameterized on the MFD size \(k\). While it performs well for small \(k\) values, it quickly becomes unfeasible for larger \(k\) values, since the parameterized runtime grows exponentially on \(k\) with a degree 2 polynomial in the exponent. This is a limiting factor to the usability of such algorithms in practical applications.

In a recent work, Dias et al. [4] formulated an Integer Linear Problem (ILP) model for MFD on DAGs and showed that it performs faster on graphs with larger solution sizes than the FPT algorithm of Kloster et al. [11]. Due to this better performance of an ILP, it is natural to work on further optimizations. An additional strength of ILP formulations is their extensibility, since ILPs can easily be modified to handle other aspects of the applications, such as subpath constraints and inexact flows cited above, see [4].
Contributions. In this paper, we significantly optimize the ILP for MFD using the notion of \textit{safe paths} for all flow decompositions in DAGs, first studied by Khan et al. \cite{10}. More specifically, a path \( P \) is said to be \textit{safe} if all flow decompositions (of any size) have at least one path containing \( P \) as a subpath. Clearly, safe paths must appear in any MFD. We show how they can be used to reduce the ILP search space by using them to fix many binary variables of the ILP. At a high level, this approach is along the same lines as for other NP-hard problems, for example, Bumpus et al. \cite{3} analyzed \textit{c-essential vertices}, as those vertices belonging to any \( c \)-approximate solution, for some specific graph problems whose solutions are sets of vertices. Clearly, \( c \)-essential vertices also belong to all optimal solutions, and can be simply removed from the graph when running e.g. an FPT algorithm (see \cite{3}).

In our case, safe solutions are sets of \textit{paths} (not single vertices), and it is non-trivial how to use them to simplify the input graph, or how to integrate them into a combinatorial algorithm (e.g., we cannot simply remove them, they could overlap, etc.). However, incorporating safe paths into an ILP is easily supported since they can be modelled by additional constraints, or by fixing some ILP variables. Moreover, in order to use more than one safe path, we observe that pairwise unreachable paths (they are not subpaths of a superpath) must be present in different MFD solution paths. As a fast heuristic to select a set of such paths of large total length, we use a reduction to a maximum weight antichain problem \cite{12}. In Appendix B we show that a \textit{maximum-length} set of pairwise unreachable safe paths can also be found in polynomial time, but this procedure is overall computationally more involved, and thus may be too expensive as a pre-processing step.

Furthermore, as observed by Cáceres et al. \cite{17}, the size of any edge antichain (like ours) is a lower bound on the MFD size. As such, we can also use this lower bound to check if a heuristic MFD solution size attains this lower bound, in which case it is optimal. To the best of our knowledge, this is the first time such a (non-trivial) lower bound is used in an MFD solver. Lastly, before running the ILP solver, we also simplify the input graph using the \( Y \)-to-\( V \) graph compaction, also used in the implementation of the FPT algorithm of Kloster et al. \cite{11}, but not used in the one of the ILP by Dias et al. \cite{4}. Using all these optimizations, we experimentally show that we obtain significant speedups over the original ILP, of up to \( 34 \times \) on the harder inputs.

In addition, we also adapt these optimizations to the MFD variants considering inexact flows and subpath constraints, proving that such adaptations can easily be done also for MFD variants of practical interest. Furthermore, we also show (in Section 4) how the \( Y \)-to-\( V \) graph compaction can be modified for these two variants, an issue which has not been considered before in the literature. With these optimizations, on the hardest graphs among the instances with inexact flows, we obtain an impressive speedup of \( 219 \times \) compared to the original ILP for MFD with subpath constraints.

As a last contribution, we tackle an MFD variant where the weights of the solution paths must belong to a given set, initially defined by Kloster et al. \cite{11}. We show that this problem on DAGs admits a simpler and thus faster ILP than the one for MFD. When the set of given weights consists of powers of 2 (up to the maximum flow value of any edge), this ILP provides a \( \log ||f|| \)-factor approximation ratio for the original MFD problem. For practical applications, we show that if we also add the flow values of all edges to the given set of weights, then this ILP returns an MFD solution with a number of paths that is significantly closer to the minimum one than the state-of-the-art heuristic MFD solver \texttt{Catfish} \cite{20}, on instances with large MFD size. These are of particular interest, since our optimized solver runs in under 2 seconds on instances of solution size at most 10. The ILP for this problem variant is up to a further order of magnitude faster than our optimized ILP, and running in under 1 second on average.
2 Preliminaries

Definitions. We let $G = (V, E)$ be a graph and assume throughout the paper that $G$ is a DAG (i.e. directed and acyclic). For simplicity we also assume that $G$ has no parallel edges and that it has a single source node $s$ and a single sink node $t$. We denote by $\mathbb{N}$ the natural numbers including 0 and by $\mathbb{Z}^+ = \mathbb{N} \setminus \{0\}$ the positive integers. A $u$-$v$ path $P$ is a sequence of edges going from the node $u$ to $v$ and its length $|P|$ is defined as its number of edges. Additionally, we identify paths $P$ with functions $P : E \to \{0, 1\}$ where $P(e) = 1$ if and only if $e \in P$. We call two paths $P_1, P_2$ (and resp. edges) independent, if there exists no path $P^*$ such that $P_1$ and $P_2$ are subpaths of $P^*$. The motivation for this definition is the fact that reachability between paths is closed under transitivity. If the DAG is a poset (i.e., $E$ is closed under transitivity), then two edges $(u, v)$ and $(x, y)$ are independent if and only if the nodes $v$ and $x$ are independent or $u$ and $y$ are independent. For a path $P = (e_1, \ldots, e_{|P|})$ we denote by $P[\ell, r] := (e_\ell, \ldots, e_r)$ a subpath of length $r - \ell + 1$. A flow network is the tuple $G = (V, E, f)$, where $f : E \to \mathbb{Z}^+$ is a flow, i.e., a function that preserves the flow conservation $\sum_{(u, v) \in E} f_{uv} = \sum_{(v, w) \in E} f_{vw}, \forall v \in V \setminus \{s, t\}$.

By a $k$-flow decomposition of a flow $f$ (see Figure 1) we denote a family of $s$-$t$ paths $\mathcal{P} = (P_1, \ldots, P_k)$ with their associated weights $w = (w_1, \ldots, w_k) \in (\mathbb{Z}^+)^k$, such that:

$$\sum_{i \in \{1, \ldots, k\}} P_i(u, v)w_i = f_{uv}, \ \forall (u, v) \in E. \quad (1)$$

In this paper, we focus on positive integer flow values, which is motivated by its application in RNA sequence reconstruction. Vatlinen et al. [24] proved that there is always a $(|E| - |V| + 2)$-flow decomposition. The Minimum Flow Decomposition (MFD) problem asks for a $k$-flow decomposition with minimum $k$.

In case of imperfect data, Williams et al. [25] considered the problem variant Minimum Inexact Flow Decomposition (MIFD) where for each edge $(u, v)$ we have a lower bound $L_{uv}$ and an upper bound $R_{uv}$, and constraint (1) is changed to require that $\sum_{i \in \{1, \ldots, k\}} P_i(u, v)w_i \in [L_{uv}, R_{uv}], \forall (u, v) \in E$. Another MFD variant of practical interest is Minimum Flow Decomposition with Subpath Constraints (MFDSC) [26]. The only difference with respect to MFD is that in the input we also have a set of paths $\mathcal{R}$, called subpath constraints, and it is required that every path in $\mathcal{R}$ is a subpath of at least one path of the decomposition.

(a) A flow network. (b) A flow decomposition into paths of weights $(1, 1, 2, 3, 3)$.

Figure 1 Example of a flow network in (1a) and a decomposition of it into five $s$-$t$ paths in (1b).

ILP formulations. We now review the ILP formulation for MFD given by Dias et al. [4], which decomposes the flow into $k$ paths. Since the value for $k$ is not known beforehand, one must iterate and increase this value until the ILP becomes feasible. To model each path $P_i$, $i \in \{1, \ldots, k\}$, binary variables $x_{uvw}$ are introduced to represent each edge $(u, v) \in E$. We set
\( x_{uv} = P(u,v) \), i.e. \( x_{uv} = 1 \) if \( (u,v) \in P \) and otherwise \( x_{uv} = 0 \). At the same time, each path is required to start from the source and end in the sink. For all the intermediary nodes of a path, a unit in-degree and a unit out-degree is required. Those requirements can be modeled by the following constraints. For all \( v \in V \) and for all \( i \in \{1, \ldots, k\} \):

\[
\sum_{(u,v) \in E} x_{uv} - \sum_{(v,u) \in E} x_{vu} = \begin{cases} 0, & \text{if } v \in V \setminus \{s,t\}, \\ 1, & \text{if } v = t, \\ -1, & \text{if } v = s, \end{cases}
\]

(2)

Constraint (1) is then modeled as:

\[
f_{uv} - \sum_{i \in \{1, \ldots, k\}} x_{uv}w_i = 0, \quad \forall (u,v) \in E.
\]

(3)

The above constraint contains the non-linear term \( x_{uv}w_i \) that standard linear solvers cannot solve. However, using basic linearization techniques, each such constraint can be replaced by three linear constraints, see [4] for details. For completeness, we give the full formulation in Appendix A.1.

Dias et al. [4] used the flexibility of this ILP formulation to easily model also the two variants MIFD and MFDSC. The former problem can easily be modeled in ILP by changing the (linearized version of) constraint (3) to state that \( \sum_{i \in \{1, \ldots, k\}} x_{uv}w_i \) belongs to the interval \([L_{uv}, R_{uv}]\). The latter problem can be modeled by introducing additional indicator variables \( r_{ij} \in \{0, 1\} \) for every \( i \in \{1, \ldots, k\} \), and every path \( R_j \in R \), modeling if the subpath \( j \) is contained in the \( i \)-th path of the flow decomposition, via the additional constraints:

\[
\sum_{(u,v) \in R_j} x_{uv} \geq |R_j|r_{ij}, \quad \forall i \in \{1, \ldots, k\}, \forall R_j \in R,
\]

(4)

\[
\sum_{i \in \{1, \ldots, k\}} r_{ij} \geq 1, \quad \forall R_j \in R.
\]

(5)

### 3 Reducing the ILP search space via safe paths

In this section, we optimize the ILP model for MFD presented above based on the notion of solution safety (see e.g. Tomescu and Medvedev [23]). Then, in Remarks 2 and 3, we discuss how this method can also be applied to the MIFD and MFDSC problems.

We call a path safe if it is part of every (not necessarily optimal) flow decomposition; that is, if it is a subpath of some path in every decomposition (not necessarily of minimum size). Khan et al. [10] characterized all safe paths via the notion of excess flow, which can be thought of as the flow value of the first edge of the path minus the flow values of the non-path edges out-going from the internal nodes of the path (i.e., the flow “leaking” from the path). Formally:

\[ f_P := f(e_1) - \sum_{e_i \neq u_{i+1}} f(u_i, v). \]

A path \( P \) is safe if and only if \( f_P > 0 \).

\[ \binom{P}{2} \]

\[ \binom{E}{2} \]

\[ \binom{V}{2} \]

\[ \binom{R}{2} \]

\[ \binom{P}{2} \]

\[ \binom{E}{2} \]

\[ \binom{V}{2} \]

\[ \binom{R}{2} \]

\[ \binom{P}{2} \]

\[ \binom{E}{2} \]

\[ \binom{V}{2} \]

\[ \binom{R}{2} \]
Since we assume the flow to be positive, every edge must be decomposed, and is thus (as a path of length 1) safe. Khan et al. [10] gave a simple algorithm for finding all safe paths, as follows. We can find all safe subpaths in each s-t path \( P = (e_1, \ldots, e_{|P|}) \) of an arbitrary flow decomposition, since safe paths are part of all flow decompositions, using a two-pointer algorithm. Starting with \( \ell = 1 \) and \( r = 2 \) as an inclusive interval (i.e., including two edges), we increase \( r \) as long as \( f_{P[\ell..r]} > 0 \), and increase \( \ell \) as long as \( f_{P[\ell..r]} \leq 0 \). That is, for every index \( r = 2, \ldots, |P| \), we find the minimum \( \ell < r \) such that the subpath \( P[\ell..r] \) is safe. The runtime of this procedure is \( O(\text{out-degree}(P)) \), where out-degree(\( P \)) := \( \sum_{(u,v) \in E} \text{out-degree}(u) \), since flows of edges across \( P \) are at most added and subtracted twice throughout the algorithm. To find the set \( S \) of all safe paths of \( G \), we can find any flow decomposition \( P \) and run the two-pointer algorithm on all paths in \( P \). For quickly obtaining a flow decomposition, one could e.g. use the greedy approach of removing paths of largest currently available flow in runtime \( O(|P| \cdot (n + m)) \) and use it as upper bound to the optimal solution in the ILP [24]. In our implementations for the three problems MFD, MIFD, MFDSC we use flow decompositions obtained from the fastest heuristic solvers for these problems.

An (edge) antichain is a subset \( Q \subseteq E \) such that every pair of edges \( e_1, e_2 \in Q \) is independent. The size of any edge antichain is also a lower bound on the size of an MFD, as noted by [17], since each edge in the antichain must be traversed by different paths in any flow decomposition by the definition of independent edges. The idea is to use such edges (and extend)

\[ \text{(a) A} \]

\[ \text{Original Safe paths. Each one in a different colour.} \]

\[ \text{(c) Maximum Weight Antichain} \]

\[ \text{Maximum Antichain Path assigned to anti chain} \]

\[ \text{(d) Long collection of independent safe paths.} \]

\[ \text{Figure 2} \]

Application of safe paths in introducing new constraints into previous ILP models. A flow network is displayed in Figure 2a. From this network, all maximal safe paths (that cannot be extended left or right) can be calculated (displayed in different colours with their respective lengths in Figure 2b). We attach to every edge the maximum length of all safe paths crossing that edge as weight. By calculating the maximum weight antichain (Figure 2c), we obtain a set of pairwise independent safe paths (Figure 2d). In the ILP formulation, we set to 1 the \( x_{uv} \) variables of each edge \((u,v)\) in the \( i\)-th safe path of the maximum weight antichain.

For every edge \( e \in E \), denote by \( a(e) \) the length of the longest safe path traversing \( e \), i.e. \( a(e) := \max_{P \in S, e \in P} |P| \); this can be obtained from the computation of safe paths described above. Let \( P(Q) = \{P_1, \ldots, P_{|Q|}\} \) be the corresponding longest safe paths passing through the edges in \( Q \); see Figure 2 for an illustration. If \( Q \subseteq E \) is an antichain, then also
the corresponding longest safe paths for the edges of $Q$ must be fully traversed by different paths in any flow decomposition (otherwise, if some $P_i$ and $P_j$ were traversed by the same $s$-$t$ path in some flow decomposition, then also the corresponding edges in $Q$ could be traversed by the same path, contradicting the fact that $Q$ is an antichain).

As such, without loss of generality, we can fix in the ILP the $i$-th path of the MFD to contain the $i$-th safe path in $\mathcal{P}(Q)$:

$$x_{uvi} = 1, \quad \forall (u, v) \in P_i, \forall P_i \in \mathcal{P}(Q). \quad (6)$$

Additionally, in order to further optimize the choice of the antichain $Q$, we consider the following notion. Given a weight function $a : E \rightarrow \mathbb{N}$, a maximum weight antichain is an antichain $Q$ maximizing $\sum_{e \in E} a(e)$ [12]. Maximum weight antichains can be found by a reduction to minimum flows, with demands on the edges given by their weights, in runtime $O(m \cdot \sum_{e \in E} a(e)) \subseteq O(n \cdot m^2)$, followed by a depth-first search through the graph. Maximum weight antichains are cut-sets with edges whose minimum flow is equal to their demand [12]. In order to heuristically fix as many $x_{uvi}$ variables as possible, we set the weight $a(e)$ of each edge $e$ to be the length of the longest safe path traversing $e$. The overall runtime of this preprocessing is thus $O(m \cdot \sum_{e \in E} a(e))$. Note that this approach does not yield the maximum total length of independent safe paths, we describe an approach on how to find them in Appendix B.

As further optimization, note that symmetries are well known to slow down ILP solvers. In the previous ILP formulations, the paths can be arbitrarily permuted, which we can mitigate with the following constraints: $w_{i+1} \leq w_i, \forall i \in \{|Q| + 1, \ldots, k - 1\}$. We show the complete optimized ILP in Appendix A.2.

Moreover, running the heuristic solvers on the instances beforehand to calculate the safe paths gives us an upper bound to the optimum solution. In our optimized ILPs we have an additional check on whether the lower bound $|Q|$ equals this upperbound. If this holds, then the heuristic flow decomposition is of minimum size, and we directly report it without running the ILP. Thus, safe paths give us the advantage, that they either yield a good lower bound, showing that heuristic methods solved the instance optimally, or can be used to accelerate the ILP when the lower bound is smaller than the solution of the heuristic solution.

**Remark 2.** Since any flow decomposition satisfying the subpath constraints is (trivially) a flow decomposition, it follows that safe paths for all flow decompositions are also safe for flow decompositions with subpath constraints. However, they do not capture all the safe paths for MFDSC. Nevertheless, we can still use such paths with positive excess flow as described above to again reduce the search space of the ILP solver for MFDSC.

**Remark 3.** Since in the MIFD problem we are given intervals, instead of a single value, for every edge, the excess flow characterization needs to be adapted. In fact, it is an open problem how to find all the safe paths for all feasible inexact flow decompositions (of any size). However, in this paper we consider a “conservative” adaptation of the excess flow notion (inexact excess flow), and prove that paths having positive inexact flow are safe for all inexact flow decompositions. As such, we can use them in the same manner as described above to reduce the search space of the ILP solver for MIFD.

**Lemma 4 (Inexact excess flow).** For a path $P = (e_1, \ldots, e_{|P|})$ with $e_i = (u_i, u_{i+1})$ define its inexact excess flow as

$$f_P := L_{e_1} - \sum_{(u_i,v) \in E, i \in \{2, \ldots, |P|\}, v \neq u_{i+1}} R_{u_i,v}.$$

A path $P$ is safe if $f_P > 0$. 

---

**A. Grigorjew, F. H. C. Dias, A. Cracco, R. Rizzi, and A. I. Tomescu**
14:8 Accelerating ILP Solvers for Minimum Flow Decompositions

Let \( f_P > 0 \) and assume that no path of a flow decomposition \( P \) of assigned weights \( w \in \mathbb{Z}_{+}^{\#P} \) routed through edge \( e_1 \) passes through all of \( P \). This means they all leave \( P \) through edges \( \text{OUT} := \{(u, v) \in E \mid i \in \{2, \ldots, k\}, v \neq u_{i+1}\} \) carrying weight at most \( \sum_{e \in \text{OUT}} R_e < L_{e_1} \). This contradicts the assumption that \( \sum_{P_j \in P, P_j(e_1) = 1} w_j \in [L_{e_1}, R_{e_1}] \).

Note that we lose the \textit{if and only if} property by assuming the worst case of only sending the lowest possible amount of flow through the first edge and of removing the largest possible amount of flow through the outgoing edges of the path. This means that inexact excess flows will not necessarily find all safe paths.

4 Y-to-V reduction for subpath constraints and inexact flows

Koster et al. [11, Lemma 4.1] used optimization to reduce the graph size in a pre-processing step by suitably contracting all edges entering nodes with in-degree 1, and respectively, exiting from nodes with out-degree 1, as we review below. This reduction has sometimes been used, under the name “Y-to-V”, to simplify the input graphs to other problems, see e.g. [23]. Note that the ILP solver by Dias et al. [4] did not include pre-processing step. In this section, we show how to extent this pre-processing step for the two problem variants MIFD and MFSDC.

The Y-to-V reduction for MFD works the following way. Consider the graph \((V, E)\) and let \( v \in V \) be a node of in-degree 1, and let \( u \in V \) be the unique node with \((u, v) \in E\). Let \( v \) have out-degree \( \ell \) and let \( w_i \in V, i = 1, \ldots, \ell \) be all nodes with \((v, w_i) \in E\). The reduction creates a new graph \((V \setminus \{v\}, E')\) with edges \((u, w_i)\) and flow \( f' : E' \to \mathbb{N} \) with \( f'(u, w_i) = f(v, w_i) \) for \( i = 1, \ldots, \ell \). This process can be repeated until no nodes of in-degree 1 exist anymore, while all other edges are copied to the new graph. Analogously this can be done for nodes of out-degree 1.

For the MFSDC problem (with subpath constraints), subpaths are given with respect to the original graph, and hence we need to modify them for the Y-to-V contracted graph. The induced subgraphs of nodes of in-degree (resp. out-degree) 1 define a forest of out-trees (resp. in-trees). Subpaths can potentially pass through, begin in or end in such trees (see Figure 3 for an example).

If a path completely passes through a tree, we can merely merge the edges of the tree in the path to the contracted edge. In that case, the subpath which intersects with the tree gets translated a single new edge. If a path stops inside an out-tree or begins inside an in-tree, we can not do that because we do not know which leaf the path will cross in the decomposition. In that case, the subpath which intersects with the tree gets translated to several new, parallel edges.

We first contract all out-trees and describe how to reduce the path to the contracted graph. We can follow up by contracting all in-trees analogously. We can partition the graph uniquely in maximal-size out-trees, such that no adjacent out-trees can be merged to a new out-tree. An out-tree in this partition consists of a single edge if and only if this edge will not be contracted in the Y-to-V reduction. Let the subpath \( R_j \in \mathcal{R} \) intersect the out-trees \( T_1, \ldots, T_{\ell_j} \), ending at node \( u \in T_{\ell_j} \). We can contract the path to obtain \( R'_j \) in the following way. Let the set of leaf nodes of \( T_{\ell_j} \) that can be reached from \( u \) be denoted by \( L(T_{\ell_j}, u) \).

For every tree \( T_1, \ldots, T_{\ell_j-1} \), we add an edge from the root of the tree to the leaf the path is crossing to \( R'_j \). For \( T_{\ell_j} \) we add the parallel edges \((\text{root}(T_{\ell_j}), v)\) for all \( v \in L(T_{\ell_j}, u) \). That means \( R'_j \) is a path of length \( \ell_j - 1 \) followed by some parallel edges. Let the obtained set of “subpaths” \( R'_j \) be \( \mathcal{R}' \). We change the constraints (4) and (5) to:
Original graph with one subpath constraint

Y-to-V contracted graph with subpath constraint, which consists of a path of length 1 followed by two edges in the last out-tree. Constraint (7) with $\ell_j = 2$ enforces that exactly one of the edges will be used.

Figure 3 A DAG with subpath constraint in green and three out-trees, which are the induced subgraphs of $\{a, b, d\}$, $\{a, c, d\}$ and $\{d, e, f, g, h, i\}$. The subpath constraint passes two out-trees in (a), and is extended to start in an out-tree root to two out-tree leaf nodes in (b).

\[
\sum_{(u,v) \in R'_j} x_{uv} \geq |\ell_j| r_{ij}, \quad \forall i \in \{1, \ldots, k\}, \forall R'_j \in \mathcal{R'},
\]

(7)

\[
\sum_{i \in \{1, \ldots, k\}} r_{ij} \geq 1, \quad \forall R'_j \in \mathcal{R'}.
\]

(8)

Note that only the set $\mathcal{R}$ and the integers $|R_j|$ change compared to the constraints (4) and (5).

Lemma 5. The constraints (7) and (8) are true in the original graph if and only if the constraints (4) and (5) are true in the Y-to-V contracted graph.

Proof. We analyze the possible routes a path $R_j$ can be extended to, such that it begins and stops at nodes that lie in between different out-trees of the graph partition.

The path $R_j$ begins in the out-tree $T_1$, potentially not at the root. Since $T_1$ is an out-tree, it can uniquely be extended to the root w.l.o.g., and we can assume that it starts at the root of $T_1$.

As such, the path passes through the out-trees $T_1, \ldots, T_{\ell_j-1}$ from the out-tree root to one of its leaves, which will be the root of the next tree. For these trees, there exists a unique edge in the contracted graph, that connects the root to that leaf.

We now analyze the path reduction for the last out-tree $T_{\ell_j}$. $R'_j$ has been constructed to contain $\ell_j - 1$ edges that form a path in the Y-to-V contracted graph, followed by parallel edges adjacent to that path. These parallel edges describe exactly all possibilities for $R_j$ to be extended to the right, such that it ends outside of the out-tree. Since they are parallel, and the variables $x_{uv}$ are constrained to represent paths, exactly one of the parallel edges is forced to be used by constraint (7).

The flow conservation is a necessary property for the Y-to-V reduction to work, since because of it, a single out-edge carries no further information. All paths of a flow decomposition entering a node of out-degree 1 must continue through this edge, and will together decompose it (and similarly for nodes of in-degree 1). In the case of MIFD, we do not require the input to have any conservation of flow, and unlike in MFD, valid inputs can also be infeasible to solve.

For nodes of in-degree 1 (resp. out-degree 1) we generalize flow conservation to the property of inexact flow conservation, which nodes must have in order for its edges to be contracted: $L_{in} \leq \sum_{(u,v) \in E} L_{uv} \leq \sum_{(u,v) \in E} R_{uv} \leq R_{in}$, where $[L_{in}, R_{in}]$ is the interval...
of the single in-edge of \( u \) (respectively for the sum of all outgoing edges and the interval \([L_{\text{out}}, R_{\text{out}}]\) of the single out-edge of \( u \)). Note that the second inequality is always fulfilled since \( L_{uv} \leq R_{uv} \). The Y-to-V reduction for MIFD works the same as the Y-to-V reduction for MFD, but defines intervals \([L'_{uw}, R'_{uw}]\) (resp. \([L'_{wv}, R'_{wv}]\)) instead of flow values \( f'(u, w_i) \) (resp. \( f'(w_i, u) \)) and is restricted on nodes of in-degree (resp. out-degree) 1 fulfilling the inexact flow conservation.

▶ **Lemma 6.** The Y-to-V reduction for MIFD is correct.

**Proof.** Consider an inexact flow decomposition of the original graph. By uniquely adapting the paths to the contracted graph, they trivially define a feasible solution, as the same set of weighted paths contribute to the same intervals as before.

Consider an inexact flow decomposition of the contracted graph. By uniquely adapting the paths to the original graph, they define a feasible solution, since the inexact flow conservation enforces the sum of the weights of paths passing through an edge with an interval that has been removed in the contracted graph to lie inside this interval. ▼

## 5 MFD with given weights

![Figure 4](image)

**Figure 4** Another flow decomposition of the flow network displayed in Fig. 1a. In this scenario, all weights are already known and limited to only power of 2 (1, 2, 4, 8) values.

In this section we consider an MFD variant in which the path weights in the solution are restricted to belong to a given set \( W = \{w_1, \ldots, w_\ell\} \), see also Figure 4 for an example and note that not all weights in \( W \) must be used by the solution paths. This problem was initially defined by Kloster et al. [11, Sec. 6], and in this paper we call it **MFD with given weights (MFDW)**. More formally, given a flow network \( G = (V, E, f) \) and a set of weights \( W = \{w_1, \ldots, w_\ell\} \subseteq \mathbb{Z}^+ \), find a minimum-sized set of \( s-t \) paths \( P = (P_1, \ldots, P_k) \) with associated weights \((\tilde{w}_1, \ldots, \tilde{w}_k)\), with each \( \tilde{w}_i \in W \), such that \( \sum_{i \in \{1, \ldots, k\}} P_i(u, v)\tilde{w}_i = f_{uv}, \forall (u, v) \in E \).

This problem has a smaller search space than MFD because the weights are restricted to \( W \), and thus can potentially admit faster solvers. Moreover, if the weights of an optimal solution are already known, then an optimal solution to MFDW is also an optimal solution to MFD, and this problem can be used as potentially faster solver. For instance, Kloster et al. [11] have observed that path weights can be found in the flow in graphs used for RNA sequence reconstruction.

In the rest of this section we show that knowing such a set \( W \), we are able to formulate a model with substantially fewer variables than the ILP model for MFD, potentially decreasing the runtime of the ILP solver by a substantial amount. This formulation also has the potential to be an alternative efficient heuristic algorithm for MFD, if \( W \) is well chosen.
Given the set of possible weights, our ILP model will be able to answer the following question: In an optimal solution (that minimizes the number of used paths), how many paths of weight \( w_i \) pass through an edge \( e \)? In other words, to decompose a flow \( f \) using the weights in \( W \), we aim to find flows \( X_i \) such that \( f = \sum_{i=1}^{W} w_i X_i \), where each \( X_i(e) \) answers that question. The ILP model for MFDW is the following:

\[
\begin{align*}
\text{Minimize} & \quad k := \sum_{(s,u) \in E} \sum_{i=1}^{[W]} x_{sui} \\
\text{Subject to:} & \quad f_{uv} = \sum_{i=1}^{[W]} w_i x_{uvi}, \quad \forall (u,v) \in E, \quad (9a) \\
& \quad \sum_{(u,v) \in E} x_{uvi} - \sum_{(v,u) \in E} x_{vui} = 0, \quad \forall v \in V \setminus \{s,t\}, \forall i \in \{1, \ldots, [W]\}, \quad (9b) \\
& \quad x_{uvi} \in \mathbb{N}, \quad \forall (u,v) \in E, \forall i \in \{1, \ldots, [W]\}. \quad (9c)
\end{align*}
\]

Note that the flow \( X_i(u, v) = x_{uvi} \) can be decomposed trivially into weight 1 paths. In addition, the product \( x_{uv} w_i \) does not require linearization, due to \( w_i \) being an input in this problem.

\begin{lemma}
The ILP model described by constraints \((9a)\) to \((9c)\) solves MFDW optimally.
\end{lemma}

\begin{proof}
Let \( \mathcal{P} = \{P_1, \ldots, P_k\} \) with weights in \( W = \{w_1, \ldots, w_{\ell}\} \) be a flow decomposition. Let \( \mathcal{P}(w_i) \subseteq \mathcal{P} \) denote the multiset of paths of weight \( w_i \in W \) and let \( x_{uvi} = |\mathcal{P}(w_i) \cap \{s\text{-}t \text{ paths crossing } (u, v)\}|. \) The variables \( x_{uvi} \) define a feasible solution to the ILP model and we have \( \sum_{(s,u) \in E} \sum_{i=1}^{[W]} x_{sui} = k \) for the objective function.

Given a feasible solution of the ILP model consisting of variables \( x_{uvi} \), construct \( s\text{-}t \) paths \( P \) of weight \( w_i \in W \) in the following way. Start at node \( s \), choose an arbitrary neighbour \( u \) of \( s \) with \( x_{sui} > 0 \) and append the edge \( (s, u) \) to \( P \). Reduce \( x_{sui} \) by one, and continue this step with \( u \) instead of \( s \) until the current node is \( t \), which finishes the path \( P \). Repeating this process until all the \( x_{uvi} \) are zero yields a flow decomposition using \( \sum_{(s,u) \in E} \sum_{i=1}^{[W]} x_{sui} \) many paths.

We now further argue that MFDW is strongly NP-hard and that solving MFDW of \( G \) with weights \( \{2^i \mid i = 0, \ldots, \lfloor \log \|f\| \rfloor \} \) optimally is a \( \lfloor \log \|f\| \rfloor + 1 \)-approximation of MFD. This is in particular useful when no small set \( W \) of optimal solution path weights is known. Depending on the application, weight sets other than powers-of-two can give a better approximation or faster runtime. We can for example use powers of 2\(^{i} \) for any \( i \in \mathbb{Z}^+ \), which yields an approximation scheme with approximation factor \((2^i - 1)[(\log \|f\| + 1)/i]\). Increasing \( i \), we expect a decreased runtime due to the reduced dimension of the ILP model, but the size of the decomposition increases.

\begin{lemma}
Let \( W = \{w_1, \ldots, w_{\ell}\} \) be a set of weights, and \( \mathcal{P} \) with associated weights \( W' = \{w'_1, \ldots, w'_{\ell}\} \) be an optimal solution of an MFD instance \( G = (V, E, f) \). If \( W' \subseteq W \), then solving MFDW with weight set \( W \) optimally solves MFD optimally.
\end{lemma}

\begin{proof}
This follows from the fact that an optimal solution of MFDW always contains at least as many paths as an optimal solution of MFD.
\end{proof}

\begin{corollary}
MFDW is strongly NP-hard.
\end{corollary}
14:12  Accelerating ILP Solvers for Minimum Flow Decompositions

Proof. Hartman et al. have shown that MFD is NP-hard on graphs using flow values only from \{1, 2, 4\} [9]. MFDW with \( W = \{1, \ldots, c\} \) solves MFD on instances where the flows are upper bounded by a constant \( c \) by Lemma 8. ◀

▶ Lemma 10. For an MFD instance \( G = (V, E, f) \), solving MFDW of \( G \) with weights \( \{2^j \mid j = 0, \ldots, \lceil \log ||f|| \rceil \} \) optimally is a \( \lfloor \log ||f|| \rfloor + 1 \)-approximation of MFD.

More generally, solving MFDW of \( G \) with weights \( \{(2^i)^j \mid j = 0, \ldots, \lceil \log ||f||/i \rceil \} \) for some \( i \in \mathbb{Z}^+ \) is a \( (2^i - 1)(\lfloor \log ||f|| \rfloor + 1)/i \)-approximation of MFD.

Proof. Let \( P = \{P_1, \ldots, P_k\} \) with associated weights \( \{w_1, \ldots, w_k\} \) be an optimal solution to \( G \) for MFD and let \( P_w \) be an optimal solution for MFDW. We can construct a feasible solution for MFDW using \( P \) by copying every \( P_i \in P \) for every positive \( j \)-th bit in the power of two decomposition of \( w_i \) and assigning weight \( 2^j \) to that copy. Since \( w_i \leq ||f|| \), this yields at most \( \lfloor \log ||f|| \rfloor + 1 \) paths for every \( P_i \in P \). Thus, \( |P_w| \leq k|\log ||f|| + 1| \).

If we only use every \( i \)-th bit, the bits on positions divisible by \( i \) need to cover the next \( i - 1 \) bits. Let the path \( P \) have weight \( w = \sum_{j=0}^{\lfloor \log ||f|| \rfloor} b_j 2^j \) for \( b_j \in \{0, 1\} \), then \( w_i = \sum_{j=0}^{\lfloor \log ||f||/i \rfloor} b_j 2^j \cdot \left( 2^i \right) \) for \( b_j \in \{0, 1\} \). The sums \( b_j + 2b_{j+1} + \cdots + 2^{i-1}b_{j+i-1} \leq 2^i - 1 \) denote how often we need to copy \( P \) using weight \( 2^i \). ◀

6 Experimental results

Solvers and datasets. Our implementation of the ILPs uses the GUROBI Python API under default settings and is available at https://github.com/algbio/optimized-fd. We compare the ILPs for MFD, MFDSC and MIFD also to the heuristic algorithms for them by [20] (Catfish), by [26] (CoasterHeuristic) and by [25] (IFDSolver), respectively. For MFDW we used the weight set \( W = \{2^i \mid i \leq \log ||f|| \} \cup \{f(e) \mid e \in E\} \).

We also experimented with the implementation from the Catfish solver of the standard greedy algorithm for MFD [24], but since we observed that it performs worse than the Catfish heuristic algorithm, we did not include it in the results. We also did not include in the results the Toboggan implementation by Kloster et al. [11] of the FPT algorithm for MFD, nor the Coaster implementation by Williams et al. [26] of the FPT algorithm for MFDSC since it was already observed in [4] that they do not scale for minimum flow decomposition sizes larger than 6 (which we also confirmed experimentally).

The runtimes of all ILPs (except MFDW) include a linear scan in increasing order to find the smallest \( k \) for which there is a flow decomposition in \( k \) paths. As discussed in Section 3, the size of the maximum weight antichain \( Q \) is a lower bound on \( k \), thus the linear scans for MFD-optimized, MFDSC-optimized and MIFD-optimized start at \( |Q| \). We set up a time limit of up to 30 minutes for each input and each method. Our experiments were performed in a server running Linux with one AMD Ryzen Threadripper PRO 3975WX 32-core CPU with 512 GB RAM.

For the MFD problem, we experimented with the datasets created by Shao and Kingsford [20]. These contain graphs created from human transcriptome using the quantification tool Salmon [15], and also datasets created from human and mouse transcriptomes using Flux-Simulator [7]. First, we took a single sample file from their dataset, referred to as SRR020730 Salmon (corresponding to the file rnaseq/salmon/sparse_quant_SRR020730.graph), followed by the entire archive present in the directories rnaseq/human/ and rnaseq/mouse/, referred to as Catfish Human and Catfish Mouse, respectively. From those two directories, only the graphs that had 50 nodes...
or more (prior to Y-to-V) were considered. For MFDSC, two datasets were used: an adaptation of the SRR020730 Salmon, where subpaths were generated and another dataset SRR30790 StringTie, which was created by Khan et al. [10] from human RNA-seq reads SRR307903 assembled using the StringTie tool [16]. In both datasets, instances are limited to four subpaths (due to performance limitations regarding previous tools used as benchmarks). Finally, for MIFD, we simulate interval flows similar to what was done in [25]: since infeasible instances are generated in the process, we repeat the process until only feasible instances are generated.

Experimental setup. To show the behaviour of the solvers for graphs of increasingly large MFD, we group the input graphs in ranges based on their MFD size, computed with MFD-optimized. If MFD-optimized does not finish within the time limit on an instance, we exclude it also from all other solvers – note that only the heuristic solvers are faster than MFD-optimized, which are not optimal in general. For each solver, we report the average runtime per range of graphs (column Avg.), and the total runtime in that range on the graphs (column Total), both in seconds, and only for the graphs on which that solver finished within the time limit. Thus, these numbers are an underestimation of the time one would need to run the solver in practice, since it would run for at least 30 minutes on unsolved instances. In column #Solved we list the number of instances on which the solver finished. We captured the runtime with the GNU time tool by separating the graphs into individual files and running the tools separately on each instance.

Column $\Delta(|P|)$ shows the approximation accuracy of the heuristic methods as follows: for each instance, we compute the difference between the number of paths reported by each formulation and the minimum number of paths in an MFD (computed with MFD-optimized). In each table cell, corresponding to a specific range of inputs, we list the sum of these differences, and in parentheses their averages.

Discussion. Table 1 illustrates the performance of the MFD solvers. MFD-optimized improves over MFD-original in all ranges of MFD size. Generally, the larger the MFD size, the more significant the runtime improvement. For example, for Catfish Human ($\min k \geq 16$), MFD-optimized is $27-31 \times$ faster than MFD-original on average and for Catfish Mouse ($\min k \geq 21$), MFD-optimized is $23 \times$ faster than MFD-original on average.

As mentioned above, in the reported runtimes of the solvers we are not including instances which did not finish within the time limit. Thus, in practice MFD-optimized has even larger speedups compared to MFD-original, since MFD-optimized solves more instances within the time limit than MFD-original, for Catfish Human and Mouse this happened for $\min k \geq 6$. In SRR020730 Salmon and Catfish Mouse MFD-optimized solves all instances, and on Catfish Human MFD-optimized solves 23 more instances.

Among the two heuristic solvers, Catfish finishes on all instances in a few seconds in total, whereas MFDW is slower than Catfish, but still running in under one second on average per graph and faster than MFD-optimized in almost all cases. When comparing the approximation accuracy, note that the instances of practical interest are those for which the MFD size is more than 10, since for smaller sizes MFD-optimized already returns an optimal solution in under 2 seconds on average. On these, MFDW is more accurate than Catfish, and interestingly, the larger the MFD size, the more accurate MFD-optimized becomes (for $\min k \geq 21$ being fully accurate on all datasets). One reason might be that in more complex graphs, the optimal MFD weights appear among the flow values of the edges (which are added to $W$).
Table 1 Results for Problem MFD. MFD-original denotes the original ILP for MFD from Dias et al. [4] (Section 2), MFD-optimized denotes our optimized ILP described in Section 3 and Section 4, and MFDW the ILP from Section 5. Runtimes are in seconds; a timeout of 30 minutes was used. The total number of instances in the datasets is mentioned in the rows “All”, in parentheses. Since the heuristic solvers finish on all instances, we do not have #Solver columns for them.

<table>
<thead>
<tr>
<th>Catfish</th>
<th>MFDW heuristic</th>
<th>MFD-optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg.</td>
<td>Total</td>
<td>∆(</td>
</tr>
<tr>
<td>1-5</td>
<td>0.01</td>
<td>38.38</td>
</tr>
<tr>
<td>6-10</td>
<td>0.01</td>
<td>19.80</td>
</tr>
<tr>
<td>11-15</td>
<td>0.01</td>
<td>6.04</td>
</tr>
<tr>
<td>16-20</td>
<td>0.01</td>
<td>0.00</td>
</tr>
<tr>
<td>21-max</td>
<td>0.01</td>
<td>0.00</td>
</tr>
<tr>
<td>All (40870)</td>
<td>0.01</td>
<td>403.99</td>
</tr>
</tbody>
</table>

In Table 2, we show the results for the MFDSC problem. We observe a similar and even more pronounced trend as for the MFD solvers, where the larger the MFDSC size, the larger the improvement of our optimized ILP. For example, for dataset SRR020730 Salmon, MFDSC-optimized is 11 ×, 69 × and 121 × faster than MFDSC-original for min k in ranges 11-15, 16-20, 21-max, respectively. For both datasets, MFDSC-optimized manages to solve one more instance than MFDSC-original, and in SRR020730 Salmon it solves all instances. Moreover, the total running time of CoasterHeuristic is more than half of the running time of MFDSC-optimized, while not giving optimal solutions (which are also less accurate on average than the heuristic solvers for MFD).

<table>
<thead>
<tr>
<th>Catfish</th>
<th>MFDSC-original</th>
<th>MFDSC-optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg.</td>
<td>Total</td>
<td>∆(</td>
</tr>
<tr>
<td>1-5</td>
<td>0.00</td>
<td>1.07</td>
</tr>
<tr>
<td>6-10</td>
<td>0.00</td>
<td>2.42</td>
</tr>
<tr>
<td>11-15</td>
<td>0.01</td>
<td>1.33</td>
</tr>
<tr>
<td>16-20</td>
<td>0.01</td>
<td>0.00</td>
</tr>
<tr>
<td>21-max</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>All (11730)</td>
<td>0.00</td>
<td>6.15</td>
</tr>
</tbody>
</table>

In Table 3, we show the results for the MIFD problem and again observe a significant speedup compared to MIFD-original on complex instances. For example, for min k in 11-15, MIFD-optimized is up to 30× faster on SRR020730 Salmon, up to 219× faster on Catfish.
Human and up to 100× faster on Catfish Mouse. On the latter two datasets, MIFD-optimized drastically runs in less than 1 second on average. In addition, although IFDSolver has, on average, a better runtime, its difference compared to MIFD-optimized is negligible. Notably, MIFD-optimized solved all instances.

### Table 3

Results for Problem MFD. Runtimes are in seconds. A timeout of 30 minutes was used.

<table>
<thead>
<tr>
<th>IFDSolver heuristic</th>
<th>MIFD-original</th>
<th>MIFD-optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>min k</td>
<td>Avg. Total</td>
<td>Avg. Total #Solved</td>
</tr>
<tr>
<td>1-5</td>
<td>0.25</td>
<td>9561.44 2161 (0.06)</td>
</tr>
<tr>
<td>6-10</td>
<td>0.26</td>
<td>512.92 1097 (0.85)</td>
</tr>
<tr>
<td>11-15</td>
<td>0.29</td>
<td>46.58 302 (1.88)</td>
</tr>
<tr>
<td>16-20</td>
<td>0.32</td>
<td>4.12 46 (3.54)</td>
</tr>
<tr>
<td>21-max</td>
<td>0.44</td>
<td>1.74 21 (5.25)</td>
</tr>
<tr>
<td>All (40870)</td>
<td>0.25</td>
<td>10126.80 4227 (0.10)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IFDSolver heuristic</th>
<th>MIFD-original</th>
<th>MIFD-optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>min k</td>
<td>Avg. Total</td>
<td>Avg. Total #Solved</td>
</tr>
<tr>
<td>1-5</td>
<td>0.24</td>
<td>2652.76 957 (0.09)</td>
</tr>
<tr>
<td>6-10</td>
<td>0.26</td>
<td>40.52 199 (1.25)</td>
</tr>
<tr>
<td>11-15</td>
<td>0.24</td>
<td>0.98 14 (3.50)</td>
</tr>
<tr>
<td>All (11073)</td>
<td>0.24</td>
<td>2694.26 1169 (0.11)</td>
</tr>
</tbody>
</table>

Finally, we computed quantitative statistics on the contributions of our various optimizations, see Table 4. Overall, the Y-to-V reduction shrinks the number of edges to less than 50%, even in the inexact flow instances where not every node of in- or outdegree 1 can be contracted. Moreover, the amount of path indicators variables set to 1 by safe paths in an antichain goes up to 21%, and is at least 10% in datasets where less than half of the instances were solved optimally using only heuristic methods (i.e., where the lower bound is equal to the upper bound). The last column shows that the safety predicted about 40% of the final solution paths has been predicted using the independent safe paths.

### 7 Conclusion

In this paper, we proposed optimizing the ILP formulations for MFD using the notion of safe paths for all flow decompositions. Since safe paths cannot be simply be removed from the graph, we observed that we can use a set of independent safe path to suitably fix a large number of ILP variables corresponding to their edges. Combined with the Y-to-V reduction, and the first usage of an antichain lower bound in a solver to detect the optimality of a heuristic solution, this resulted in a significantly faster MFD solver (up to 34× on the harder instances). We also developed an ILP that can work as a heuristic for MFD, running in under 1 second on average, and more accurate than the state-of-the-art heuristic Catfish on graphs of practical interest. We also showed that these optimizations can be applied to two MFD variants of practical interest (by also adapting the Y-to-V reduction the first time for these variants), resulting in even bigger speedups.

Future research encompasses extending such improvements to the ILPs for MFD in general graphs with cycles [5]. Another point of improvement is a further search space reduction through further optimizations on the application of safety, for example, by answering
Table 4: Quantitative statistics of the various optimisations. The first two columns |E| original and |E| Y-to-V show the number of edges in the original and in the Y-to-V contracted graph (and, in parentheses, the proportion of this number relative to the original graphs), respectively, summed over all graphs in that dataset. The column LB = UB shows the proportion of instances in each dataset where the calculated lower bound (the number of safe paths in a maximum antichain) is equal to the upper bound (the number of heuristically calculated paths). This column indicates in how many instances we have proven that the heuristic solver is optimum using safety. The column \( p_{IND} \) shows the proportion of all path indicators \( x_{uv} \) variables that were set to 1 as part of a safe path with respect to all edges of the contracted graph, where \( i \) is the index of the path and \( (u, v) \) is an edge. The last column \( \#IND/\sum |P| \) shows the proportion of path indicator variables with respect to the sums of the solution's path lengths, since the total solution path length is an upperbound on how many variables we can set to 1.

|                | |E| original | |E| Y-to-V | LB = UB | \( p_{IND} \) | \( \#IND/\sum |P| \) |
|----------------|-----------------|----------|------|--------|----------|-----------------|-----------------|
| **MFD**        |                  |          |      |        |          |                 |                 |
| SRR020730 Salmon | 354519          | 147996   | 32.32% | 10.95% | 48.60%   |                 |                 |
| Catfish Human  | 621297           | 109870   | 63.67% | 9.03%  | 41.85%   |                 |                 |
| Catfish Mouse  | 687763           | 126388   | 54.89% | 8.36%  | 37.37%   |                 |                 |
| **MFDSC**      |                  |          |      |        |          |                 |                 |
| SRR020730 Salmon | 214380          | 94986    | 91.45% | 8.05%  | 43.71%   |                 |                 |
| SRR30790 StringTie | 24912          | 8926     | 31.02% | 15.84% | 48.63%   |                 |                 |
| **MIFD**       |                  |          |      |        |          |                 |                 |
| SRR020730 Salmon | 355526          | 166136   | 26.77% | 11.91% | 23.39%   |                 |                 |
| Catfish Human  | 115774           | 48903    | 34.12% | 20.46% | 27.29%   |                 |                 |
| Catfish Mouse  | 101123           | 40865    | 28.05% | 21.92% | 28.02%   |                 |                 |

the following question: Is it tractably possible to find subpaths that are part of all \( \alpha \)-approximations of MFD, where \( \alpha \) can either be a constant or depend on the input size? Such \( \alpha \)-safe paths (following the notion of \( c \)-essential vertices by Bumpus et al. [3]) would be longer the closer \( \alpha \) is to 1, while safe paths in this paper would refer to \( \infty \)-safe paths. For the problem variants MFDSC and MIFD, can we find all \( \infty \)-safe paths in polynomial time?

References


Accelerating ILP Solvers for Minimum Flow Decompositions

A.1 MFD-original

\[
\forall i \in \{1, \ldots, k\}:
\sum_{(s,v) \in E} x_{svi} = 1 \quad (10a)
\sum_{(u,t) \in E} x_{uti} = 1 \quad (10b)
\sum_{(u,v) \in E} x_{uvi} - \sum_{(v,w) \in E} x_{vwi} = 0 \quad \forall v \in V \setminus \{s,t\}, \quad (10c)
\]

\[
f_{uv} = \sum_{i \in \{1, \ldots, k\}} \phi_{uvi} \quad \forall (u,v) \in E, \quad (10d)
\]

\[
\forall (u,v) \in E, \forall i \in \{1, \ldots, k\}:
\phi_{uvi} \leq f_{\text{max}} x_{uvi} \quad (10e)
\phi_{uvi} \leq w_i \quad (10f)
\phi_{uvi} \geq w_i - (1 - x_{uvi}) f_{\text{max}} \quad \forall i \in \{1, \ldots, k\}, \quad (10g)
\]

\[
w_i \in Z^+ \quad \forall i \in \{1, \ldots, k\}, \quad (10h)
\]

\[
x_{uvi} \in \{0, 1\} \quad \forall (u,v) \in E, \forall i \in \{1, \ldots, k\}, \quad (10i)
\]

\[
\phi_{uvi} \in \mathbb{N} \quad \forall (u,v) \in E, \forall i \in \{1, \ldots, k\}. \quad (10j)
\]

A.2 MFD-optimized

Equations (10) together with:

\[
x_{uvi} = 1, \quad \forall (u,v) \in P_i, \forall P_i \in \mathcal{P}(Q) \quad (11a)
\]

\[
w_{i+1} \leq w_i, \quad \forall i \in \{|Q|+1, \ldots, k-1\} \quad (11b)
\]

A.3 MFDW

\[
\text{Minimize} \quad \sum_{(s,u) \in E} \sum_{i=1}^{\lfloor W \rfloor} x_{sui}
\]
Subject to:

\[ f_{uv} = \sum_{i=1}^{\lfloor W \rfloor} w_i x_{uvi}, \quad \forall (u,v) \in E \] (12a)

\[ \sum_{(u,v) \in E} x_{uvi} - \sum_{(v,u) \in E} x_{vui} = 0, \quad \forall v \in V \setminus \{s,t\}, \forall i \in \{1, \ldots, \lfloor W \rfloor\} \] (12b)

\[ x_{uvi} \in \mathbb{N}, \quad \forall (u,v) \in E, \forall i \in \{1, \ldots, \lfloor W \rfloor\} \] (12c)

### B Maximum length independent safe paths

The approach used in Section 3 to calculate independent safe paths of a flow network \((G,f)\) does not return them of maximum length, but works only as a heuristic. In order to find independent safe paths of maximum length in polynomial time, one can, as in the heuristic approach, reduce to maximum weight antichains, on the following dependency graph \(D = (V(D), E(D))\): nodes represent safe paths, with given weights of their length. A directed edge \((u,v)\) is added if there exists a path in \(G\) which first traverses both safe paths represented by \(u\) and \(v\), entering the safe path of \(u\) first and then entering the safe path of \(v\).

▶ **Lemma 11.** A \(A \subseteq V(D)\) is an independent set in \(D\) if and only if the set of safe paths corresponding to the nodes in \(A\) is independent in \(G\), where the total weight of \(A\) is the total length of the corresponding safe paths. As a result, a maximum length independent path set of safe paths in \(G\) can be calculated in polynomial time \(O(|S|^2 \cdot \sum_{S \in \mathcal{S}} |S|) \subseteq O(|S|^2 \cdot n \cdot m)\), where \(S\) is the set of all safe paths.

**Proof.** The graph \(D\) is a transitive graph, i.e. \((u,v), (v,w) \in E(D)\) implies that \((u,w) \in E(D)\). That is because path dependencies are transitive: If there is a path \(P_1\) traversing safe paths \(s_1\) and \(s_2\), and there is a path traversing safe paths \(s_2\) and \(s_3\), one can construct a path that traverses all of \(s_1, s_2, s_3\). Thus, independent node sets in \(D\) correspond exactly to independent safe paths in \(G\): Two safe paths are independent if and only if they are not connected by an edge in \(D\). The graph \(D\) can be constructed in \(O(|S|^2 \cdot m)\) time, by performing a graph search for every pair of safe paths to check whether they are independent or not.

It is a classical result ([13]), that finding a maximum weighted independent node set on transitive graphs is solvable in polynomial time by finding a maximum weight node antichain. Since the weights of the nodes are \(|S|\) for \(S \in \mathcal{S}\), it runs in time \(O(|E(D)| \cdot \sum_{S \in \mathcal{S}} |S|) \subseteq O(|S|^2 \cdot \sum_{S \in \mathcal{S}} |S|)\).
Determining Fixed-Length Paths in Directed and Undirected Edge-Weighted Graphs

Daniel Hambly
School of Mathematics, Cardiff University, Wales, UK

Rhyd Lewis
School of Mathematics, Cardiff University, Wales, UK

Padraig Corcoran
School of Computer Science and Informatics, Cardiff University, Wales, UK

Abstract

In this paper, we examine the NP-hard problem of identifying fixed-length s-t paths in edge-weighted graphs – that is, a path of a desired length \(k\) from a source vertex \(s\) to a target vertex \(t\). Many existing strategies look at paths whose lengths are determined by the number of edges in the path. We, however, look at the length of the path as the sum of the edge weights. Here, three exact algorithms for this problem are proposed: the first based on an integer programming (IP) formulation, the second a backtracking algorithm, and the third based on an extension of Yen’s algorithm. Analysis of these algorithms on random graphs shows that the backtracking algorithm performs best on smaller values of \(k\), whilst the IP is preferable for larger values of \(k\).

2012 ACM Subject Classification Theory of computation → Backtracking; Theory of computation → Integer programming; Information systems → Fixed length attributes

Keywords and phrases Graphs, paths, backtracking, integer programming, Yen’s algorithm

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.15


Funding Daniel Hambly: EPSRC

1 Introduction

In this paper, we consider the problem of finding fixed-length paths in directed and undirected edge-weighted graphs. That is, we want to find a path in a network from one point to another where the total length of the path, in terms of the sum of its edge weights, is as close to possible as some prescribed length \(k\). A particular application of interest is that of exercise. A street network can be conceptualised as a graph, where intersections and dead ends act as vertices, and roads as edges. Designing a workout route for activities like walking, running, or cycling, involves determining a path from a starting point to a destination within a specified number of steps, calories to be burned, or distance to cover. The selection of criteria, such as steps, calories, or distance, would define the weights assigned to the edges, making the task akin to identifying a path of a fixed length. Routing services are a feasible option for finding a solution, although they usually focus on routes with minimum length rather than a specific length [28]. Fixed-length paths can also be used in the combinatorial optimisation problem of determining a fixed-length cycle. Adding a dummy vertex to a graph, with edges and edge weights equal to those that leave the source vertex, setting it as the target vertex, and finding a path from the source to the target would be equivalent to finding a fixed-length cycle from the source. This is applicable for determining exercise routes for fixed-length
Determining Fixed-Length Paths in Directed and Undirected Edge-Weighted Graphs

circuits and cycles [28, 29, 40]. Further applications of fixed-length paths include identifying interaction pathways between proteins [24], and detecting signaling pathways in protein interaction networks [34].

We now review some definitions and results relevant to this work:

Definition 1. In a graph, a walk is a sequence of vertices and edges; a trail is a sequence of vertices with no repeated edges; and a path is a trail with no repeated vertices.

To indicate that a walk/trail/path starts at a vertex $u$ and ends at a vertex $v$, the prefix $u-v$ is added to the terms above. In the case when $u=v$, the following terms are used:

Definition 2. A $u-v$ walk/trail/path is called closed if $u=v$, with closed trails and paths labelled as circuits and cycles respectively.

Similarly to before, a $u$-circuit (cycle) can be used to describe a circuit (cycle) that starts and ends at a vertex $u$.

We can now formally define the $k$-length path problem (KPP):

Definition 3. Let $G = (V,E)$ be a directed or undirected edge-weighted graph with $n$ vertices and $m$ edges, let $k \geq 0$ be a desired length, and let $w(u,v)$ be a nonnegative weight for each edge $\{u,v\} \in E$. The $k$-length path problem involves determining a path $P = (v_1,v_2,...,v_l)$ from a source vertex $s = v_1$ to a target vertex $t = v_l$ in $G$ such that its length $L(P) = \sum_{i=1}^{l-1} w(v_i,v_{i+1})$ minimises $|k - L(P)|$.

The problem of determining the shortest or longest $s$-$t$ paths is also included in our definition by setting $k = 0$ or $k \geq \sum_{\forall \{u,v\} \in E} w(u,v)$ respectively.

In the next section, we will look at some bounds and existing solutions for finding fixed-length paths in graphs. In Section 3, we then establish three different methods for the KPP, whilst Section 4 analyses the performance of these algorithms. Finally, Section 5 concludes the paper and discusses ideas for future work.

2 Literature Review

Basagni et al. [2] show that determining a walk of length $k$ in directed and undirected edge-weighted graphs is an NP-complete problem, though it can be solved in polynomial time if the graph is unweighted and $k = n^{O(1)}$. The number of $k$-length walks in unweighted graphs can be found by taking the $k$th exponent of the binary adjacency matrix of the graph [26]; specifically, the number of $k$-length $s$-$t$ walks, where the length is equal to the number of edges, is equal to $(A^k)_{ij}$ [39]. Eppstein [14] showed that the $K$ shortest trails can be found in $O(m + n \log(n) + K)$ time using a variation of Yen’s algorithm (discussed in greater detail in Subsection 3.3).

The shortest path problem is a well-studied problem that is known to be solvable in polynomial time with a variety of different algorithms [36]. The related problem of finding the longest $s$-$t$ path in a graph is known to be $\mathcal{NP}$-hard [36] and fixed-parameter tractable on unweighted graphs [8], though it can be solved in polynomial time on directed acyclic graphs [35]. The KPP is a generalisation of the longest path problem when $k$ is large enough and is, therefore, also $\mathcal{NP}$-hard. Moreover, akin to the longest path problem, the KPP can be solved in polynomial time on trees and directed acyclic graphs [23]. Further, Williams [41] notes that the KPP can be solved in polynomial time when $k \leq \log(n)/\log(\log(n))$. The problems of enumerating the total number of $k$-length paths or $s$-$t$ paths are $\#W[1]$-complete [15] and $\#P$-complete [38] respectively.
Various bounds on the lengths of paths are noted in the literature. A trivial lower bound on the length of an \( s-t \) path in an edge-weighted graph is to find the shortest \( s-t \) path. This can be computed in polynomial time, with the time complexity of the algorithm dependent on the algorithm chosen. A theorem from Dirac [13] states that, if \( \deg(v) \geq d \ \forall v \in V \) for an undirected unweighted graph \( G \) and \( d \geq 0 \), then there is a path of length at least \( d \) in \( G \). This was extended by Bondy and Fan [9] to edge-weighted graphs to form the following theorem where \( \deg^w(u) = \sum_{v \in \Gamma(u)} w(u, v) \), and \( \Gamma(u) \) is the neighbourhood of \( u \).

**Theorem 4.** Let \( G = (V, E) \) be a connected undirected edge-weighted graph, \( u \in V \), and \( d \geq 0 \). If \( \deg^w(v) \geq d \ \forall v \in V \setminus \{u\} \) and \( w(u, v) \geq 0 \ \forall \{u, v\} \in E \), then \( G \) contains a path from \( u \) with length at least \( d \).

Gallai et al. [20] theorise that, for a biconnected undirected unweighted graph \( G \) with \( u, v \in V \) and \( d \geq 0 \), if \( \deg(x) \geq d \ \forall x \in V \setminus \{u, v\} \), then there is a \( u-v \) path of length at least \( d \) in \( G \). Again, this was extended by Bondy and Fan [9] to edge-weighted graphs:

**Theorem 5.** Let \( G = (V, E) \) be a biconnected undirected edge-weighted graph, \( u, v \in V \), and \( d \geq 0 \). If \( \deg^w(x) \geq d \ \forall x \in V \setminus \{u, v\} \), then \( G \) contains a \( u-v \) path with length at least \( d \).

Frieze et al. [19] have proved a conjecture from [9] that provides a further lower bound on the length of a path:

**Theorem 6.** Let \( G = (V, E) \) be an undirected edge-weighted graph. Then \( G \) contains a path with length at least \( \frac{2}{n} \sum_{(u,v) \in E} w(u,v) \).

Fomin et al. [17] have also broadened the results from [13] and [20] to obtain the following corollary and theorem respectively:

**Corollary 7.** Let \( G = (V, E) \) be a connected graph, \( B \subseteq V, \ k \geq 0 \), and let \( \delta(G) \) denote the minimum vertex degree \( \forall v \in V \). Deciding whether \( G \) contains a path of length at least \( \min\{2\delta(G - B), |V| - |B| - 1\} + k \) is solvable in \( 2^{O(k+|B|)}n^{O(1)} \) time.

**Theorem 8.** Let \( G = (V, E) \) be a biconnected graph, \( B \subseteq V, \ s, t \in V, \ k \geq 0 \), and let \( \delta(G) \) denote the minimum vertex degree \( \forall v \in V \). Deciding whether \( G \) contains an \( s-t \) path of length at least \( \delta(G - B) + k \) is solvable in \( 2^{O(k+|B|)}n^{O(1)} \) time.

Several upper bounds for the length of the longest path are trivial to compute. For example, in an undirected edge-weighted graph \( G \) where \( e_1, ..., e_n \) are the \( n \) edges with the largest weights in \( G \), an upper bound is \( \sum_{i=1}^{n} w(e_i) \). Lewis et al. [29] further provide an upper bound, of \( \left( \sum_{v \in V} \tau(v) \right)/2 \), for the length of the longest cycle in a biconnected undirected edge-weighted graph \( G \). Here, \( \tau(v) \) signifies the sum of the weights of the two heaviest edges connected to a vertex \( v \). This also serves as an upper bound for the longest path in the graph.

In terms of solution strategies for the KPP and related problems, Monien [30] was amongst the first to provide an algorithm that could determine if a path between two vertices has a length equal to \( k \). The algorithm constructs a matrix \( D^{(k)}(G) \), where each entry \( d_{ij}^{(k)} \) for \( i, j \in V \) is either equal to \( k \) if a path of length \( k \) between \( i \) and \( j \) exists, or \( \lambda \) otherwise. However, the construction of the matrix takes exponential time \( O(k!nm) \) to compute. Bodlaender [8] has shown that there exists an algorithm that can prove the existence of a path or an \( s-t \) path of length \( \geq k \) in \( O(2^k k!n) \) and \( O(2^{2k}(2k)!n + m) \) time respectively. Nonetheless, these algorithms are only feasible in terms of run-time for very
small values of \( k \). Bax et al. [4] created a finite-difference method to count the number of \( k \)-length \( s-t \) paths in directed graphs. However, this algorithm also runs in exponential time \( O(2^k \text{poly}(n)) \). Papadimitriou et al. [32] initially conjectured the possibility of finding paths with lengths \( \log(n) \) in polynomial time. This was subsequently validated by Alon et al. [1], who additionally introduced a colour-coding method to identify \( k \)-length paths in \( O(2^k m \log(n)) \) and \( O(2^k n \log(n)) \) time for directed and undirected unweighted graphs, respectively.

Kneis et al. [25] introduced the divide-and-colour method, combining divide-and-conquer and colour-coding techniques to locate paths of \( k \) vertices in unweighted graphs. Operating with a time complexity of \( O(3^{\log(k)} k^k) \) and an exponentially small error probability, this approximation algorithm also featured a deterministic counterpart capable of finding \( k \)-vertex paths in \( O(4^k k^2 \log^2(n)) \) time. Around the same time, Chen et al. [12] proposed a divide-and-conquer approximation algorithm and a deterministic algorithm for this problem in unweighted graphs, with runtimes of \( O(4^k k^{3.42} m) \) and \( O(12.8^k mn) \), respectively. Chen et al. [11] extended this research to achieve an approximation and deterministic algorithm of \( O(4^k k^{2.7} m) \) and \( O(4^k + O((\log(k))^{3}) nm) \) time on undirected edge-weighted graphs, respectively. Koutis [27] introduced an \( O^*(2^{3k/2}) \) approximation algorithm for this problem on directed unweighted graphs, which was then expanded to achieve an \( O^*(2^k) \) approximation algorithm [41]. Björklund et al. [7] realised the first approximation algorithm with an exponent base below \( 2 \) for determining the existence of \( k \)-vertex paths in undirected graphs in \( O^*(1.66^k) \) time, whilst Fomin et al. [18] developed a deterministic algorithm on directed and undirected graphs with time complexities of \( O(2.619^k n \log(n)) \) and \( O(2.619^k m \log(n)) \), respectively. Zehavi [43] improved upon this by introducing an \( O^*(2.597^k) \) time deterministic algorithm that uses divide-and-colour to find \( k \)-vertex paths with length at least \( W \) on directed edge-weighted graphs. This work was then extended to accomplish an \( O^*(2.554^k) \) time deterministic algorithm for determining \( k \)-vertex \( s-t \) paths on directed unweighted graphs [37]. Fomin et al. [16] then also presented a \( 2^k n^{O(1)} \) randomised algorithm for the same problem on undirected unweighted graphs.

In other work, Bax [3] employed inclusion-exclusion algorithms to enumerate all paths and \( s-t \) paths in graphs. However, both algorithms exhibit the same exponential runtimes as observed in [4]. Roberts and Kroese [33], meanwhile, utilised a Monte Carlo simulation to estimate the number of \( s-t \) paths in both directed and undirected unweighted graphs, whilst Björklund et al. [6] introduced an algorithm applying a linear transformation on a function, enabling the computation of the total number of \( s-t \) paths in directed edge-weighted graphs based on both length and the number of vertices \( 0 \leq k \leq n - 1 \) in the path. The latter algorithm runs in \( O^*(\exp(H(k/2n)n)) \) time, where \( H(p) = -p \log(p) - (1 - p) \log(1 - p) \) and \( 0 \leq p \leq 1 \). Similarly, Birmelé et al. [5] devised an algorithm listing all \( s-t \) paths in undirected unweighted connected graphs in \( O(m + \sum_{\pi \in P_{st}(G)} |\pi|) \) time. Here, \( P_{st}(G) \) represents the set of all \( s-t \) paths in \( G \), \( \pi \) is an \( s-t \) path, and \( |\pi| \) is the number of edges in \( \pi \). A different approach, due to Montanari et al. [31], used Markov chains to sample \( s-t \) paths in undirected unweighted planar graphs, considering path length as the number of edges. Longer paths have a small acceptance probability, whilst shorter paths are always accepted. More recently, a general purpose algorithm was introduced, capable of finding all paths up to length \( k \) in \( O(n + m + (k^2 + \Delta)|S_k|) \) time on a directed edge-weighted graph \( G \), where \( k \) is a specified number of vertices, \( \omega \) is the exponent of matrix multiplication, \( \Delta \) is the maximum degree vertex in \( G \), and \( |S_k| \) is the number of connected induced subgraphs [21]. This algorithm is known to be fixed-parameter tractable if \( |S_k| = O(f(k)\text{poly}(n)) \).
Figure 1 An example of adding a dummy vertex. Part (a) shows the graph and the source and target vertices \( s \) and \( t \) respectively. Part (b) shows the addition of the dummy vertex \( d \) with edges to \( s \) and \( t \). The dashed edges show a possible \( d \)-cycle. Part (c) shows the \( s-t \) path formed from the \( d \)-cycle with \( d \) removed.

3 Proposed Solutions

Most of the methods reviewed in Section 2 considered unweighted graphs in which the length of the path is defined as the number of edges it contains. For this paper, we look at edge-weighted graphs and consider the length of the path as the sum of the edge weights. Over the course of the upcoming subsections, we will examine three different exact methods to solve the KPP – that is, given excess time, they guarantee the optimal solution. The first is an IP formulation, of which we believe is the first for determining \( k \)-length paths. The second is a backtracking algorithm that has an in-built pruning mechanism for reducing the size of the solution space. This is an improvement on similar techniques noted in Section 2 as, unless \( k \) is close in length to that of the longest paths, not all \( s-t \) paths need to be considered to find the optimal solution. The final is an extension of Yen’s algorithm.

3.1 Integer Programming (IP) Formulation

In this subsection, we introduce an IP formulation for the KPP. This formulation is adapted from a model for finding fixed-length cycles in undirected edge-weighted graphs [29]. Specifically, we introduce a dummy vertex \( d \) with edges to \( s \) and \( t \) and edge weights equal to 0. By running the aforementioned model to find a fixed-length cycle from \( d \), we effectively achieve the task of determining a fixed-length path from \( s \) to \( t \). Figure 1 gives an example of how adding a dummy vertex to the graph can work.

For a graph \( G = (V, E) \), let \( V = \{v_0, v_1, v_2, ..., v_n\} \), and set \( v_0 \) as the dummy vertex \( d \) with edges to \( s \) and \( t \) and edge weights equal to 0. In the following formulation, the parameters \( A_{n+1 \times n+1} \) and \( W_{n+1 \times n+1} \), and the variable \( X_{n+1 \times n+1} \) denote the adjacency matrix, the weight matrix, and the binary decision variables respectively. \( A_{ij} = 1 \) if \( \{v_i, v_j\} \in E \) and 0 otherwise, \( W_{ij} = w(v_i, v_j) \) if \( \{v_i, v_j\} \in E \) and \( \infty \) otherwise, whilst \( X_{ij} = 1 \) if there is an edge from \( v_i \) to \( v_j \) in the \( d \)-cycle, and 0 otherwise. The objective is to therefore minimise \( |k - \sum_{i=0}^n \sum_{j=0}^n A_{ij}W_{ij}X_{ij}| \). As this is not linear, we introduce a variable \( Z \) and Constraints (1) and (2) to linearise it. The full model is to now minimise \( Z \) subject to:
Determine Fixed-Length Paths in Directed and Undirected Edge-Weighted Graphs

\[ k = \sum_{i=0}^{n} \sum_{j=0}^{n} A_{ij} W_{ij} X_{ij} \leq Z \]  

(1)

\[ \sum_{i=0}^{n} \sum_{j=0}^{n} A_{ij} W_{ij} X_{ij} - k \leq Z \]  

(2)

\[ X_{ij} \leq A_{ij} \quad \forall i \in \{0, \ldots, n\}, \forall j \in \{0, \ldots, n\} \]  

(3)

\[ \sum_{i=0}^{n} X_{ij} - \sum_{i=0}^{n} X_{ji} = 0 \quad \forall j \in \{0, \ldots, n\} \]  

(4)

\[ \sum_{i=0}^{n} X_{ij} + \sum_{i=0}^{n} X_{ji} \leq 2 \quad \forall j \in \{0, \ldots, n\} \]  

(5)

\[ \sum_{i=0}^{n} \sum_{j=0}^{n} A_{ij} X_{ij} \geq 3 \]  

(6)

\[ 0 \leq Y_{ij} \leq n X_{ij} \quad \forall i \in \{1, \ldots, n\}, \forall j \in \{0, \ldots, n\} \]  

(7)

\[ 2 \sum_{j=0}^{n} Y_{ij} - 2 \sum_{j=1}^{n} Y_{ji} - \sum_{j=1}^{n} X_{ij} - \sum_{j=1}^{n} X_{ji} = 0 \quad \forall i \in \{1, \ldots, n\} \]  

(8)

\[ A_{ds}, A_{sd}, A_{dt}, A_{td}, X_{ds}, X_{td} = 1 \]  

(9)

\[ X_{ij} \in \{0,1\} \quad \forall i \in \{0, \ldots, n\}, \forall j \in \{0, \ldots, n\} \]  

(10)

\[ Y_{ij}, Z \in \mathbb{R} \quad \forall i \in \{0, \ldots, n\}, \forall j \in \{0, \ldots, n\} \]  

(11)

In this formulation, Constraint (3) stipulates that an edge is eligible to be part of the cycle only if its corresponding entry in the adjacency matrix is set to 1. Constraints (4) and (5) ensure a vertex is only in the \(d\)-cycle if it has one edge both entering and leaving. Constraint (6) ensures the cycle must contain a minimum of three edges (the shortest cycle possible is \((d,s,t,d)\), which contains three edges). Due to the structure of the graphs generated in this paper and Constraints (4) and (9), a \(d\)-cycle will always contain at least 3 edges, rendering Constraint (6) redundant. We have still included it here, however, for completeness. Constraints (7) and (8) incorporate the auxiliary variables \(Y_{ij}\) to ensure that the solution consists of only a single \(d\)-cycle. A proof of this can be found in [10]. Finally, Constraint (9) simply adds edges, in both directions, from \(d\) to \(s\) and \(t\). It also guarantees that the edges \{\(d, s\)\} and \{\(t, d\)\} are present in the cycle, since we want the path to start at \(s\) and end at \(t\).

3.2 Backtracking Algorithm

Our second exact method for the KPP is a backtracking algorithm. This is based on an extension to depth-first search that can enumerate all \(s-t\) paths. Initially, the neighbourhood \(\Gamma(s)\) is added to a list, and a vertex \(u \in \Gamma(s)\) is considered. If \(u\) is not already in the current path, it is given a label of the length of the path from \(s\) to \(u\) and added to the current path. All vertices of \(\Gamma(u)\) are then added to the back of the list and the vertices in this neighbourhood are then considered. Therefore, the neighbours of a vertex that have been added to the back of the list are considered before those preceding it. In this sense, the list is a stack. If \(t\) is considered in a neighbourhood, the \(s-t\) path and its length are output. Then, instead of adding \(t\) to the path and \(\Gamma(t)\) to the list, the algorithm considers the next vertex of the neighbourhood at the back of the list. This method continues until an \(s-t\) path of length \(k\) or all \(s-t\) paths have been found.
To shorten execution times, this algorithm also uses a cutoff value \( \bar{k} \), which represents the length of the shortest observed \( s-t \) path whose length exceeds \( k \). Initially, \( \bar{k} \) is set to \( \infty \). When an \( s-t \) path with a length greater than \( k \) but less than \( \bar{k} \) is found, \( \bar{k} \) is then updated. During execution, if an \( s-u \) path is encountered whose length exceeds \( \bar{k} \) then, due to the assumption that all edge weights are nonnegative, no extensions to this \( s-u \) path need to be evaluated. If \( \bar{k} \) assumes the value of \( k \) during execution, then an optimal solution has been found. If \( \bar{k} \) was not included in this way, the algorithm would simply find all \( s-t \) paths.

In this method, each \( s-t \) path is computed in \( O(n + m) \) time. Given a graph with \(|\mathcal{P}|\) such paths, the overall runtime is \( O(|\mathcal{P}|(n + m)) \) in the worst-case. Since \(|\mathcal{P}|\) can approach the exponential with regards to graph size, this may be infeasible for large and/or dense graphs. As the algorithm considers the neighbourhoods of a vertex at the back of the list instead of the front, DFS will start at one end of the graph and systematically work its way around the rest of the graph. If the optimal solution is one of the last \( s-t \) paths to be considered, the algorithm may run for very long periods.

### 3.3 Yen’s Algorithm

Our final approach for this problem is based on an extension of Yen’s algorithm [42], which is a method for outputting the sequence of \( K \) shortest \( s-t \) paths in length order. Upon the production of each \( s-t \) path, Yen’s algorithm operates by modifying the graph based on the previously observed paths, allowing the creation of the next path in the sequence. If Dijkstra’s algorithm is employed using a Fibonacci heap, the overall time complexity of this algorithm is \( O(Kn(m + n \log(n))) \). Here, it is necessary to set \( K = \infty \), though we can include a clause that terminates the algorithm once an \( s-t \) path with a length greater than or equal to \( k \) is found. For large graphs and cases where the length of the shortest \( s-t \) path is significantly lower than \( k \), the algorithm may need to produce a restrictively large number of paths, leading to infeasible runtimes and memory requirements.

### 4 Analysis

In this section, we analyse and compare the performance of our three approaches on a set of random Euclidean graphs. In our trials, each graph was generated by assigning \( n \) vertices to random positions in a \((10,000 \times 10,000)\)-unit square. Edges were then created between each pair of vertices with probability 0.5, with edge weights then set to the Euclidean distance between the endpoints. Our IP formulation was implemented on Gurobi version 10.0.2, and the backtracking algorithm and Yen’s algorithm were written in C++.

For the algorithms in C++, adjacency lists were used to store the graphs, whilst for the IP formulation, adjacency matrices were required. All trials were executed on Windows 10 machines with 3.5 GHz and 8.0 RAM. A time limit of 1800 seconds was used for each algorithm.

Figure 2 shows the mean runtimes of executions that terminated in less than our time limit. In all cases, the considered \( k \) values used in these charts range from zero to the longest observed \( s-t \) paths for each graph. It is clear that for these values of \( n \), Yen’s algorithm can only find the optimal solution in the given time limit for the smallest values of \( k \). This is because there are usually very many \( s-t \) paths with lengths greater than the shortest \( s-t \) path that will need to be enumerated by the algorithm before a path of sufficient length can be found. The backtracking algorithm has the shortest runtimes for most values of \( k \);
**Figure 2** Runtimes for \( n = 50 \) and 100 (respectively) on random graphs for various \( k \)-values. Each point is the mean across 20 graphs, with shaded areas indicating one standard deviation on either side.

**Table 1** Values of \( k \) where the algorithms returned optimal solutions within the time limit in all runs. Here, \( \leq \infty \) indicates that the algorithm returned solutions in all cases for all values of \( k \), including when \( k \) was larger than the length of the longest \( s-t \) path. (For \( n = 50 \) and 100 the longest paths had lengths of approximately 370,000 and 750,000 respectively).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>( n )</th>
<th>( \leq ) 50</th>
<th>( \leq ) 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
<td>( \leq \infty )</td>
<td>( \leq \infty )</td>
<td></td>
</tr>
<tr>
<td>Backtracking</td>
<td>( \leq 240,000 )</td>
<td>( \leq 450,000 )</td>
<td></td>
</tr>
<tr>
<td>Yen’s</td>
<td>( \leq 15,000 )</td>
<td>( \leq 10,000 )</td>
<td></td>
</tr>
</tbody>
</table>

However, it also features a rapid increase in run times when \( k \) exceeds a certain threshold (as detailed in Table 1), producing unpredictable runtimes and a high standard deviation. As \( k \) eventually reaches the lengths of the longest \( s-t \) paths in the graph, many of the paths found have lengths lower than the cutoff value. Therefore, the cutoff value is rarely updated and more paths are discovered. Since the total number of paths can approach the exponential, the algorithm cannot search the whole graph in the time limit. Note that, unlike Yen’s algorithm, which enumerates paths in length order, the ordering of paths observed by the backtracking algorithm is arbitrary. Hence, unlike the former, it has the potential to evaluate long paths early in the run. In addition, the cutoff value reduces the solution space and prevents costly solutions from being considered. These features, combined with its lower complexity, brings better results.

In contrast, although the IP method tends to have longer runtimes for lower values of \( k \), it also features better scaling-up properties than both other algorithms, with optimal solutions being found within the time limit over a larger range of \( k \) values. Indeed, it was able to find the optimal solutions for all tested values of \( k \).

Table 1 shows the maximum values for \( k \) for which the algorithms had 100% success rates. Yen’s algorithm had a 100% success rate for the smallest values of \( k \), with \( k \) decreasing further as \( n \) increased. The backtracking algorithm had 100% success rates on most values of \( k \), the exception being when \( k \) was close to the maximum \( s-t \) path length. In contrast, the IP had 100% success rates for all values of \( k \), with the optimal solution found in all cases. This includes the longest \( s-t \) paths in the graphs.
5 Conclusion

This paper has analysed three different exact methods on the $\mathcal{NP}$-hard problem of finding $k$-length $s$-$t$ paths in directed and undirected edge-weighted graphs. One method is an adaptation of an IP formulation on the related problem of finding $k$-length cycles, the second is a backtracking algorithm that contains a cutoff value to prevent expensive solutions from being explored, and the third is an extension of Yen’s algorithm.

The backtracking algorithm was shown to perform best on most values of $k$; however, it struggled to produce the optimal solution on higher values of $k$ in the given time limit. Yen’s algorithm could only find optimal solutions for the smallest values of $k$. In contrast, the IP was able to find the optimal solutions for all values of $k$, including when $k$ was greater than the length of the longest $s$-$t$ path.

Several avenues can be explored to further the work on the KPP. For the backtracking algorithm, adding a heuristic could help with finding $t$ earlier in the run. Studying the literature to see if any other methods could be used to compare against the IP model would be beneficial. For example, heuristics such as local search or tabu search could be considered, where many different neighbourhood operators could be employed. The inclusion of some preprocessing of the graphs before any algorithms are run would be helpful to further reduce the search space and speed up the runtimes of the algorithms. In terms of the analysis in this paper, future work could come in the form of using randomised distance for the edge-weights instead of the Euclidean distance, or extending the analysis for both directed and undirected graphs. Finally, analysing the algorithms over different graph topologies that represent real-life networks (for example, planar graphs or grid graphs) would establish supportive evidence for which algorithm operates best in a real-world setting.

References

Determining Fixed-Length Paths in Directed and Undirected Edge-Weighted Graphs


Solving the Optimal Experiment Design Problem
with Mixed-Integer Convex Methods

Deborah Hendrych
Zuse Institute Berlin, Germany
Technische Universität Berlin, Germany

Mathieu Besançon
Université Grenoble Alpes, Inria, LIG, Grenoble, France
Zuse Institute Berlin, Germany

Sebastian Pokutta
Technische Universität Berlin, Germany
Zuse Institute Berlin, Germany

Abstract
We tackle the Optimal Experiment Design Problem, which consists of choosing experiments to run
or observations to select from a finite set to estimate the parameters of a system. The objective is to
maximize some measure of information gained about the system from the observations, leading to a
convex integer optimization problem. We leverage Boscia.jl, a recent algorithmic framework, which
is based on a nonlinear branch-and-bound algorithm with node relaxations solved to approximate
optimality using Frank-Wolfe algorithms. One particular advantage of the method is its efficient
utilization of the polytope formed by the original constraints which is preserved by the method,
unlike alternative methods relying on epigraph-based formulations. We assess our method against
both generic and specialized convex mixed-integer approaches. Computational results highlight the
performance of our proposed method, especially on large and challenging instances.

2012 ACM Subject Classification Theory of computation → Branch-and-bound; Theory of computa-
tion → Integer programming; Theory of computation → Convex optimization

Keywords and phrases Mixed-Integer Non-Linear Optimization, Optimal Experiment Design, Frank-
Wolfe, Boscia

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.16


WithBoscia, archived at swh:1:dir:165e9afd84b473294aabe56781360b9d527c96b9

Funding Research reported in this paper was partially supported through the Research Campus Modal
funded by the German Federal Ministry of Education and Research (fund numbers 05M14ZAM,05M20ZBM)
and the Deutsche Forschungsgemeinschaft (DFG) through the DFG Cluster of Excellence MATH+.

1 Introduction

The Optimal Experiment Design Problem (OEDP) arises in statistical estimation and empirical
studies in many applications areas from Engineering to Chemistry. For OEDP, we assume
we have a matrix $A$ consisting of the rows $v_1, \ldots, v_m \in \mathbb{R}^n$ where each row represents an
experiment. The ultimate aim is to fit a regression model:

$$\min_{\theta \in \mathbb{R}^n} \|A\theta - y\|,$$

© Deborah Hendrych, Mathieu Besançon, and Sebastian Pokutta;
licensed under Creative Commons License CC-BY 4.0

22nd International Symposium on Experimental Algorithms (SEA 2024).
Editor: Leo Liberti; Article No. 16; pp. 16:1–16:22
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
where $y$ encodes the responses of the experiments and $\theta$ are the parameters to be estimated. The set of parameters with size $n$ is assumed to be (significantly) smaller than the number of distinct experiments $m$. Furthermore, we assume that $A$ has full column rank, i.e. the vectors $v_1, \ldots, v_m$ span $\mathbb{R}^n$.

The problem is running all experiments, potentially even multiple times to account for errors, is often not realistic because of time and cost constraints. Thus, OEDP deals with finding a subset of size $N$ of the experiments providing the “most information” about the experiment space [23, 12]. The number of allowed experiments $N$ has to be larger than $n$ to allow the regression model in Equation (1) to be solved. It is, however, assumed to be less than $m$.

In Section 2, we investigate the necessary conditions for a function to be a valid and useful information measure. Every information function leads to a different criterion. In this paper, we focus on two popular criteria, namely the A-criterion and D-criterion, see Section 2.1.

In general, OEDP leads to a Mixed-Integer Non-Linear Problem (MINLP). There has been a lot of development in the last years in solving MINLP [17]. Nevertheless, the capabilities of current MINLP solvers are far away from their linear counterparts, the Mixed-Integer Problem (MIP) solvers [4], especially concerning the magnitude of the problems that can be handled. Therefore, instead of solving the actual MINLP, a continuous version of OEDP, called the Limit Problem, is often solved and the integer solution is created from the continuous solution by some form of rounding [23]. This does not necessarily lead to optimal solutions, though, and the procedure is not always applicable to a given continuous solution either.

The goal of this paper is to compare the performance of different MINLP approaches for OEDP problems. A special focus is put on our newly proposed framework Boscia [14] which can solve larger instances and significantly outperforms the other examined approaches. Since it leverages a formulation and solution method that differs from other approaches, we establish the convergence of the Frank-Wolfe algorithm used for the continuous relaxations on the considered OEDP problems in Section 3. The different solution methods are detailed in Section 4 and the evaluation of the computational experiments can be found in Section 5.

### 1.1 Related Work

As mentioned, one established method of solution is the reduction to a simpler problem by removing the integer constraints and employing heuristics to generate an integer solution from the continuous solution. Recently, there have been more publications tackling the MINLP formulation of the Optimal Experiment Design Problem. These, however, concentrate on specific information measures, in particular the A-criterion, see [21, 1], and the D-criterion, [27, 22, 19]. The most general solution approach known to the authors was introduced in [2]. It considers OEDP under matrix means which, in particular, includes the A-criterion and D-criterion. While the matrix means covers many information measures of interest, it still yields a restricting class of information functions. For example, the G-criterion and V-criterion are not included in this class of functions [12]. Our newly proposed framework Boscia only requires the information measures to be either $L$-smooth, i.e. the gradient is Lipschitz continuous, or generalized self-concordant, thereby covering a larger group of information functions. In addition, Boscia does not suppose any prior knowledge about the structure of the problem, being thus more flexible in terms of problem formulations. On the other hand, it is highly customizable, giving the user the ability to exploit the properties of their problems to speed up the solving process. An in-depth and unified theory for the Optimal Experiment Design Problem can be found in [23].
1.2 Contribution

Our contribution can be summarized as follows.

Unified view on experiment design formulations

First, we propose a unified view of multiple experiment design formulations as the optimization of a nonlinear (not necessarily Lipschitz-smooth) information function over a truncated scaled probability simplex intersected with the set of integers. Unlike most other formulations that replace the nonlinear objective with nonlinear and/or conic constraints, we preserve the original structure of the problem. Additionally, we can easily handle special cases of OEDP without any reformulations since we do not suppose a specific problem structure, unlike the approach in [2].

Superior solution via the Boscia framework

We use the recently proposed Boscia framework [14] solving MINLPs with a Frank-Wolfe method for the node relaxations of a branch-and-bound tree, and show the effectiveness of our method on instances generated with various degrees of correlation between the parameters.

1.3 Notation

In the following let $\lambda_i(A)$ denote the $i$-th eigenvalue of matrix $A$; we assume that these are sorted in increasing order. Moreover, $\lambda_{\text{min}}(A)$ and $\lambda_{\text{max}}(A)$ denote the minimum and maximum eigenvalue of $A$, respectively. Further, let $\log\det(\cdot)$ be the log-determinant of a positive definite matrix. Given matrices $A$ and $B$ of the same dimensions, $A \circ B$ denotes their Hadamard product. Given a vector $x$, $\text{diag}(x)$ denotes the diagonal matrix with $x$ on its diagonal. The cones of positive definite and positive semi-definite matrices in $\mathbb{R}^{n \times n}$ will be denoted by $\mathbb{S}^n_{++}$ and $\mathbb{S}^n_{+}$, respectively. We will refer to them as PD and PSD cones. For $m \in \mathbb{Z}_{>0}$, let $[m] = \{1, 2, \ldots, m\}$. Lastly, we denote matrices with capital letters, e.g. $X$, vectors with bold small letters, e.g. $x$, and scalars as simple small letters, e.g. $\lambda$.

2 Optimal Experiment Design

As explained in the introduction of the paper, the matrix $A$ encodes the experiments. We are allowed to perform $N$ experiments and are interested in finding the subset with the “most” information gain. Consequently, an important question to answer is how to quantify information. To that end, we introduce the information matrix

$$X(x) = \sum_{i=1}^{m} x_i v_i v_i^T = A^T \text{diag}(x) A$$

where $x_i \in \mathbb{Z}_{\geq 0}$ denotes the number of times experiment $i$ is to be performed. Throughout this paper, we will use both ways of expressing $X(x)$ but will favor the second representation.

The inverse of the information matrix is the dispersion matrix:

$$D(x) = \left( \sum_{i=1}^{m} x_i v_i v_i^T \right)^{-1}.$$ 

It is a measure of the variance of the experiment parameters [1]. Maximizing over the information matrix is equivalent to minimizing over the dispersion matrix [23].
Note the following properties of the information matrix. The matrix $A^\top A$ has full rank and is positive definite. Because of the non-negativity of $x$, the matrix $X(x)$ is in the PSD cone. In particular, $X(x)$ is positive definite for $x \in \mathbb{Z}_m^m$ if the non-zero entries of $x$ correspond to at least $n$ linearly independent columns of $A$. Observe that the dispersion matrix only exists if $X(x)$ is positive definite, i.e. has full rank. To solve the regression problem (1), the information matrix $X(x^*)$ corresponding to the optimal solution $x^*$ of OEDP has to lie in the PD cone.

**Remark 1.** Experiments can be run only once or be allowed to run multiple times to account for errors. In the latter case, we will suppose upper (and lower bounds) on the number of times a given experiment can be run. The sum of the upper bounds usually greatly exceeds $N$. If non-trivial lower bounds $l$ are present, their sum may not exceed $N$ otherwise there is no solution respecting the time and cost constraints.

Now we answer the question posed earlier: How can we measure information? We need a function $\phi$ receiving a positive definite matrix as input and returning a number, that is $\phi : S^{n}_{++} \rightarrow \mathbb{R}$. We will lose information by compressing a matrix to a single number. Hence, the suitable choice of $\phi$ depends on the underlying problem. Nevertheless, there are some properties that any $\phi$ has to satisfy to qualify as an information measure.

**Definition 2 (Information Function [23]).** An information function $\phi$ on $S^{n}_{++}$ is a function $\phi : S^{n}_{++} \rightarrow \mathbb{R}$ that is positively homogeneous, concave, nonnegative, non-constant, upper semi-continuous and respects the Loewner ordering.

The Loewner Ordering is an ordering on the PSD cone. Let $D, B \in S^{n}_{++}$, then:

$$D \succeq B \text{ if and only if } D - B \in S^{n}_{++}.$$  

Respecting the Loewner ordering together with concavity ensures that the intuition that running more experiments should not result in less information is upheld. Scaling should not change differences in information hence we require positive homogeneity. A constant function is not useful in this setting and non-negativity is a convention. Upper semi-continuity ensures that there are no sudden jumps around the optimal objective value of the optimization problem.

The **Optimal Experiment Design Problem** can then be defined as:

$$\begin{align*}
\max_{x} & \quad \log(\phi(X(x))) \\
\text{s.t.} & \quad \sum_{i=1}^{m} x_i = N \\
& \quad 1 \leq x \leq u \\
& \quad x \in \mathbb{Z}^m,
\end{align*}$$

(OEDP)

where $u$ and $l$ denote the upper and lower bounds, respectively.

**Remark 3.** The log does not change the ordering and thus, is not strictly necessary. It can help reformulate the objective function, though.

A special case of non-trivial lower bounds is obtained if $n$ linearly independent experiments have non-zero lower bounds. These experiments can be summarized in the matrix $C = A^\top \text{diag}(l)A$. Notice that $C$ is positive definite. The information matrix then becomes:

$$X_C(x) = C + A^\top \text{diag}(x - l)A.$$
Definition 4 (Optimal Problem and Fusion Problem). In case the lower bounds are all zero, we call the resulting problem the Optimal Problem during this paper. Replacing the information matrix $X(x)$ with the fusion information matrix $X_C(x)$ in the objective in (OEDP), yields the so-called Fusion Problem.

The resulting optimization problem is an integer non-linear problem which, depending on the information function $\phi$, can be $\mathcal{NP}$-hard. The two information measures we will focus on lead to $\mathcal{NP}$-hard problems [2]. For convenience in the latter section, we define

$$P := [l, u] \cap \left\{ x \in \mathbb{R}^m_{\geq 0} \mid \sum_{i=1}^{m} x_i = N \right\}$$

(Convex Hull)

as the feasible region of the OEDPs without the integer constraints, i.e. the convex hull of all feasible integer points. Thus, the feasible region with integer constraints will be noted by $P \cap \mathbb{Z}^m$.

2.1 The A-Optimal Problem and D-Optimal Problem

The most frequently used information functions arise from the matrix means $\phi_\rho$ [23, 1].

Definition 5 (Matrix Mean). Let $B \in S^n_+$ and let $\lambda(B)$ denote its eigenvalues. The matrix mean $\phi_\rho(B)$ is defined as

$$\phi_\rho(B) = \begin{cases} 
\lambda_{\text{max}}(B), & \text{for } \rho = \infty, \\
\left(\frac{1}{n} \text{Tr}(B^p)\right)^{\frac{1}{p}}, & \text{for } \rho \neq 0, \pm\infty, \\
\det(B)^{\frac{1}{n}}, & \text{for } \rho = 0, \\
\lambda_{\text{min}}(B), & \text{for } \rho = -\infty, \\
0 & \text{for } \rho = [-\infty, 0] \text{ and } B \text{ singular.}
\end{cases}$$

(2)

Note that the matrix means function $\phi_\rho$ satisfies the requirements of Definition 2 only for $\rho \leq 1$ [23]. The two most commonly used criteria arising from matrix means are the D-optimality and A-optimality criteria, corresponding to $\rho = 0$ and $\rho = -1$, respectively. We convert the problems to a minimization form from this point on for homogeneity with the convention of the used solution methods.

2.1.1 D-Criterion

Choosing $\rho = 0$ and noting that $\log \left( \det(X)^{\frac{1}{n}} \right) = \frac{1}{n} \log \det X$, yields

$$\min_x - \log \det(X(x))$$

s.t. $x \in P \cap \mathbb{Z}^m$

(D-Opt)

as the D-Optimal Experiment Design Problem (D-Opt). Observe that the objective is equivalent to minimizing the determinant of the dispersion matrix. This determinant is also called the generalized variance of the parameter $\theta$ [23]. A maximal value of $\det X$ corresponds to a minimal volume of standard ellipsoidal confidence region of $\theta$ [22]. Additionally, the D-criterion is invariant under reparameterization, see [23].
2.1.2 A-Criterion

For parameters with a physical interpretation, the A-optimality criterion is a good choice as it amounts to minimizing the average of the variances of $\theta$ [23]. Using the log rules for the objective, the \textit{A-Optimal Experiment Design Problem} (A-Opt) can be stated as

\begin{align*}
\min_{x} & \quad \text{Tr} \left( (X(x))^{-1} \right) \\
\text{s.t.} & \quad x \in \mathcal{P} \cap \mathbb{Z}^m.
\end{align*}

(A-Opt)

Note that the D-Optimal Problem is known to be \textit{NP}-hard since the eighties [27]. \textit{NP}-hardness of the A-Optimal Problem and the D-Fusion Problem was proved only recently, see [21] and [22], respectively. The hardness of the A-Fusion Problem is still open though the authors conjecture that it too is \textit{NP}-hard.

3 Convergence guarantees for the continuous subproblems

To apply the new framework Boscia, we need to guarantee that the Frank-Wolfe algorithm converges on the continuous subproblems. The conventional property guaranteeing convergence is \textit{L-smoothness} of the objective, that is

$$\exists L \in \mathbb{R}_{>0} \text{ such that } \| \nabla f(x) - \nabla f(y) \| \leq L \| x - y \| \forall x, y \in \mathcal{P}.$$ 

In the case of both the D-Optimal Problem and the A-Optimal Problem, not every point $x \in \mathcal{P}$ is domain-feasible for the objective functions due to the corresponding information matrix being singular. For such points $x$, we define $-\log \det(X(x)) = \infty$ and $\text{Tr} \left( (X(x))^{-1} \right) = \infty$ for the D-criterion and A-criterion, respectively. Thus, the objective functions of (A-Opt) and (D-Opt) are not \textit{L}-smooth over the feasible region.

Lacking \textit{L}-smoothness on the feasible region, we require a different property that guarantees convergence of the Frank-Wolfe algorithm for the Optimal Problems under both criteria. By [7], the Frank-Wolfe algorithm also converges (with similar convergence rates) if the objective is \textit{generalized self-concordant}.

\begin{citedef}[Generalized Self-Concordance [26].]
A three-times differentiable, convex function $f : \mathbb{R}^n \to \mathbb{R}$ is $(M_f, \nu)$-generalized self-concordant with order $\nu > 0$ and constant $M_f \geq 0$, if for all $x \in \text{dom}(f)$ and $u, v \in \mathbb{R}^n$, we have

$$\left| \langle \nabla^3 f(x)[u]v, v \rangle \right| \leq M_f \|u\|_2^2 \|v\|_2^{\nu - 2} \|v\|_2^{3-\nu}$$

where

$$\|w\|_z = \langle \nabla^2 f(x)w, w \rangle$$

and $\nabla^3 f(x)[u] = \lim_{\gamma \to 0} \gamma^{-1} (\nabla^2 f(x + \gamma u) - \nabla^2 f(x))$.

Self-concordance is a special case of generalized self-concordance where $\nu = 3$ and $u = v$. This yields the condition

$$\left| \langle \nabla^3 f(x)[u]u, u \rangle \right| \leq M_f \|u\|_2^3.$$

For a univariate, three times differentiable function $f : \mathbb{R} \to \mathbb{R}$, $(M_f, \nu)$-generalized self-concordance\footnote{For self-concordance, $\nu = 3$.} condition is

$$|f'''(x)| \leq M_f f''(x)^{\frac{3}{2}}.$$
By [26, Proposition 2], the composition of a generalized self-concordant function with a linear map is still generalized self-concordant. Hence, it suffices that we show that the functions $f(X) = -\log \det(X)$ and $g(X) = \text{Tr}(X - p)$, $p > 0$, are generalized self-concordant for $X$ in the PD cone. Note that $f$ is the logarithmic barrier for the PSD cone which is known to be self-concordant [20]. Convergence of the Frank-Wolfe algorithm for the (D-Opt) Problem is therefore guaranteed.

For the (A-Opt) Problem, we can show that $g$ is self-concordant on a part of the PD cone, namely one characterized by an upper bound on the maximum eigenvalue.

\begin{theorem}
The function $g(X) = \text{Tr}(X - p)$, with $p > 0$, is $(3, \frac{\sqrt{2a(p+2)}}{p(p+1)})$-generalized self-concordant on $\text{dom}(g) = \{X \in S^n_{++} : 0 \prec X \preceq aI\}$ where $a \in \mathbb{R}_{>0}$ bounds the maximum eigenvalue of $X$.
\end{theorem}

The proof of Theorem 7 and the proof that the maximum eigenvalue of the information matrix $X(x)$ has indeed an upper bound can be found in Appendix A. Thus, we have established convergence for both the D-Optimal Problem and the A-Optimal Problem. The objectives of the A-Fusion Problem and D-Fusion Problem are generalized self-concordant by the argument. Note that the objectives for the Fusion Problems are also $L$-smooth due to their information matrix always being positive definite.

\section{Solution Methods}

The main goal of this paper is to propose a new solution method for the Optimal and Fusion Problems under the A-criterion and the D-criterion based on the novel framework Boscia and assess its performance compared to several other convex MINLP approaches. In the following, we introduce the chosen MINLP solvers and state the necessary conditions and possible reformulations that are needed. We have chosen MINLP solution approaches which require relatively few changes to the formulations in (D-Opt) and (A-Opt). Hence, we are not investigating, for example, second-order cone formulations like in [24] in the scope of this paper.

\textbf{Branch-and-Bound with Frank-Wolfe methods (Boscia)}

The new framework introduced in [14] is implemented in the Julia package Boscia.jl. It is a Branch-and-Bound (BnB) framework that utilizes Frank-Wolfe methods to solve the relaxations at the node level. The Frank-Wolfe algorithm [13, 5], also called Conditional Gradient algorithm [18], and its variants are first-order methods solving problems of the type:

$$\min_{x \in \mathcal{X}} f(x)$$

where $f$ is a convex, Lipschitz-smooth function and $\mathcal{X}$ is a compact convex set. These methods are especially useful if the linear minimization problem over $\mathcal{X}$ can be solved efficiently. The Frank-Wolfe methods used in Boscia are implemented in the Julia package FrankWolfe.jl, see [3].

At each iteration $t$, the Frank-Wolfe algorithm solves the linear minimization problem over $\mathcal{X}$ taking the current gradient as the linear objective, resulting in a vertex $v_t$ of $\mathcal{X}$. The next iterate $x_{t+1}$ is computed as a convex combination of the current iterate $x_t$ and the vertex $v_t$. Many Frank-Wolfe variants explicitly store the vertex decomposition of the iterate, henceforth called the \textit{active set}. We utilize the active set representation to facilitate warm starts in Boscia by splitting the active set when branching.
One novel aspect of Boscia is its use of a *Bounded Mixed-Integer Linear Minimization Oracle (BLMO)* as the *Linear Minimization Oracle (LMO)* in the Frank-Wolfe algorithm. The BLMO solves the mixed-integer linear problem over the feasible region \( \mathcal{X} \) with additional bound constraints. Typically, the BLMO is a MIP solver but it can also be a combinatorial solver. This leads to more expensive node evaluations but has the benefit that feasible integer points are found from the root node and the feasible region is much tighter than the continuous relaxation for many problems. In addition, Frank-Wolfe methods can be lazified, i.e. calling the LMO at each iteration in the node evaluation can be avoided, see [6].

In the case of OEDP, strong lazification is not necessary since the corresponding BLMO is very simple. The feasible region \( \mathcal{P} \) is just the scaled probability simplex intersected with integer bounds, which is amenable to efficient linear optimization. Given a linear objective \( d \), we first assign \( x = l \) to ensure that the lower bound constraints are met. Next, we traverse the objective entries in increasing order, adding to the corresponding variable the value of \( \max\{u - l, N - \sum(x)\} \). This way, we ensure that both the upper-bound constraints and the knapsack constraint are satisfied. The BLMO over the feasible set can also be cast as a simple network flow problem with \( m \) input nodes connected to a single output node, which must receive a flow of \( N \) while the edges respect the lower and upper bounds.

Due to the convexity of the objective, the difference \( \langle \nabla f(x_t), x_t - v_t \rangle \) is upper bounding the primal gap \( f(x_t) - f(x^*) \) at each iteration. We call the quantity the dual gap (or the Frank-Wolfe gap). The dual gap can therefore be used as a stopping criterion. The error adaptiveness of the Frank-Wolfe algorithm can be exploited to a) solve nodes with smaller depth with a coarser precision and b) dynamically stop a node evaluation if the lower bound on the node solution exceeds the objective value at the incumbent.

Observe that in contrast to the epigraph-based formulation approaches that generate many hyperplanes, our method works with the equivalent of a single supporting hyperplane given by the current gradient and moves this hyperplane until it achieves optimality. It is known that once the optimal solution is found, a single supporting hyperplane can be sufficient to prove optimality (as described e.g. for generalized Benders decomposition in [25]). Finding this final hyperplane, however, may require adding many cuts beforehand at suboptimal iterations. In the case of the problems discussed in this paper, the constraint polytope is uni-modular. Adding hyperplanes created from the gradient will not maintain this structure and in consequence, yields a numerically more challenging MIP. Our approach, on the other hand, keeps the polytope and thereby its uni-modularity intact.

**Outer Approximation (SCIP+OA)**

Outer Approximation schemes are a popular and well-established way of solving MINLPs [17]. This approach requires an epigraph formulation of (OEDP):

\[
\begin{align*}
\min_{t,x} & \quad t \\
\text{s.t.} & \quad t \geq -\log (\phi_p(x)) \\
& \quad x \in \mathcal{P} \\
& \quad t \in \mathbb{R}, x \in \mathbb{Z}^m. 
\end{align*}
\]

(E-OEDP)

This approach approximates the feasible region of (E-OEDP) with linear cuts derived from the gradient of the non-linear constraints, in our case \( \nabla f \). Note that this requires the information matrix \( X(x) \) at the current iterate \( x \) to be positive definite, otherwise an evaluation of the gradient is not possible or rather it will evaluate to \( \infty \). The implementation is done with the
Julia wrapper of SCIP [4]. Observe that generating cuts that prohibit points leading to singular $X(x)$, we will refer to them as domain cuts, is non-trivial. Thus, this approach can only be used for the Fusion Problem where the corresponding information matrix is always positive definite.

LP/NLP Branch-and-Bound (Pajarito)

Another Outer Approximation approach, as implemented in Pajarito.jl [11], represents the non-linearities as conic constraints. This is particularly convenient in combination with the conic interior point solver Hypatia.jl [10] as it implements the log det cone (the epigraph of the perspective function of log det) directly:

$$K_{\log \det} := \text{cl} \left\{ (u, v, W) \in \mathbb{R} \times \mathbb{R}_{>0} \times \mathbb{S}^n_{++} \mid u \leq v \log \det(W/v) \right\}.$$  

The formulation of (D-Opt) then becomes

$$\max_{t, x} t \quad \text{s.t.} \quad (t, 1, X(x)) \in K_{\log \det} \quad x \in P \quad t \in \mathbb{R}, x \in \mathbb{Z}^m.$$ 

For the representation of the trace inverse, we utilize the dual of the separable spectral function cone [9, Section 6]:

$$K_{sepspec} := \text{cl} \left\{ (u, v, w) \in \mathbb{R} \times \mathbb{R}_{>0} \times \text{int}(\mathcal{Q}) \mid u \geq v \varphi(w/v) \right\}.$$ 

For our purposes, $\mathcal{Q}$ is the PSD cone and the spectral function $\varphi$ is the negative square root whose convex conjugate is precisely the trace inverse, see [8, Table 1]²:

$$K_{sepspec}^* := \text{cl} \left\{ (u, v, w) \in \mathbb{R} \times \mathbb{R}_{>0} \times \text{int}(\mathbb{S}^n_+) \mid v \geq u/4 \text{Tr}((w/u)^{-1}) \right\}.$$ 

The conic formulation of (A-Opt) is therefore

$$\min_{t, x} 4t \quad \text{s.t.} \quad (1, t, X(x)) \in K_{sepspec}^* \quad x \in P \quad t \in \mathbb{R}, x \in \mathbb{Z}^m.$$ 

Additionally, the conic formulation allows for the computation of domain cuts for $x$. Hence, this solver can be used on all problems. Note that we use HiGHS [15] as a MIP solver within Pajarito.

A Custom Branch-and-Bound for OEDP (Co-BnB)

The most general solver strategy for (OEDP) with matrix means criteria was introduced in [2]. Like Boscia, it is a Branch-and-Bound-based approach with a first-order method to solve the node problems. The first-order method in question is a coordinate-descent-like

² For further details, see this discussion [16].
algorithm. We refer to this approach as Co-BnB in the rest of this paper. As the termination
criterion, this method exploits the fact that the objective function is a matrix mean and
shows the connection of the resulting optimization problem:

\[
\begin{align*}
\max_w \log (\phi_p(X(w))) \\
\text{s.t.} \quad \sum_{i=1}^m w_i &= 1 \\
\quad w_i &\geq 0 \\
\quad w_i N &\in \mathbb{Z} \forall i \in [m] \\
\end{align*}
\]

(M-OEDP)
to the generalization of the Minimum Volume Enclosing Ellipsoid Problem (MVEP) [2]. The
variables \( w \) can be interpreted as a probability distribution and the number of times the
experiments are to be run is denoted by \( wN \). Concerning (OEDP), one can say \( x = wN \).
In the case of the Fusion Problem, the already completed experiments have to be explicitly
added as variables with fixed values.

Note that we have improved and adapted the step size rules within the first-order method,
see the arvix version linked on the front page. Further, note that the solver was developed
for instances with a plethora of experiments and very few parameters. The solver employs
the simplest Branch-and-Bound strategy, i.e. utilizing the most fractional branching rule and
traversing the tree using the minimum lower bound. In the next section, we will see that the
method works well in cases where \( n \) is small but struggles if \( n \) increases.

5 Computational Experiments

In this section, we present the computational experiments for the Optimal Problem and
Fusion Problem, both under the A- and D-criterion, respectively. The resulting problems will
be referred to as the A-Fusion Problem (AF), D-Fusion Problem (DF), A-Optimal Problem
(AO), and D-Optimal Problem (DO).

Experimental Setup

For the instance generation, we choose the number of experiments \( m \in \{50, 60, 80, 100, 120\} \),
the number of parameters \( n \in \{\lfloor m/4 \rfloor, \lfloor m/10 \rfloor\} \), and the number of allowed experiments
\( N = \lfloor 1.5n \rfloor \) for the Optimal Problems and \( N \in [m/20, m/3] \) for the Fusion Problems. The
lower bounds are zero. Note that for the Fusion Problems, the fixed experiments are encoded
in a separate matrix. The upper bounds are randomly sampled between 1 and \( N/3 \) for
(A-Opt) and (D-Opt). In the Fusion case, they are sampled between 1 and \( m/10 \). We
generate both independent and correlated data. Also, note that the matrices generated are
dense. The number of random seeds is 5. In total, there are 50 instances for each combination
of problem and data set.

Experiments were run on a cluster equipped with Intel Xeon Gold 6338 CPUs running at
2 GHz and a one-hour time limit.

Start Solution

Note that both the objectives (D-Opt) and (A-Opt) are only well defined if the information
matrix \( X(x) \) has full rank. This is the case for the Fusion Problem, not necessarily for the
Optimal Problem. Both Boscia and Co-BnB require a feasible starting point \( z_0 \). For its
construction, we find a set \( S \subset [m] \) of \( n \) linearly independent experiments, i.e. \( n \) linearly
independent rows of $A$. Assign those experiments their upper bound. If the sum $\langle 1, z_0 \rangle$ exceeds $N$, remove 1 from the experiment with the largest entry. If the sum is less than $N$, pick an experiment in $[m]\setminus S$ at random and assign it as many runs as possible. Repeat until the sum $\langle 1, z_0 \rangle$ is equal to $N$. Note that due to the monotonic progress of both first-order methods, the current iterate will never become domain infeasible, i.e. $X(x)$ will not become singular. If the iterate is domain infeasible after branching, we discard that node.

Results

An overview of the results of the computational experiments is given in Table 1. The new framework Boscia solves the most instances by far. In comparison to the Outer Approximation methods, it terminates nearly twice as often. The Co-BnB fares better. Nonetheless, it solves fewer instances to optimality than Boscia. The notable exception is the A-Fusion Problem with correlated data. Note that in general, it fares well for the instances where $n = \lfloor m/10 \rfloor$ as it was designed for such problems. It struggles for the instances where $n = \lfloor m/4 \rfloor$.

In terms of time, Boscia also shows promising results, especially for instances of larger scale. For small-scale instances, the Outer Approximation approaches are fast, in some cases faster than Boscia. The graphical view of the number of solved instances over time is shown in Figure 1 for the Optimal Problems and in Figure 2 for the Fusion Problems. Notice that there is a sharp increase in the beginning, especially for Co-BnB and Boscia. In contrast to Boscia, the curves for Co-BnB flatten more after the first 10 seconds. The notable exception is the A-Fusion Problem with correlated data for which the Co-BnB method works consistently better than boscia, see also Table 1. The better performance of Co-BnB can be explained by the different objectives used. For the A-criterion, Co-BnB uses $\log (\text{Tr}(X^{-1}))$ instead of $\text{Tr}(X^{-1})$ which both Boscia and SCIP utilize. Computing the Optimal Problems and Fusion Problems using the logarithm formulation with both Boscia and SCIP yields twice as good a performance on the A-Fusion problem with correlated data, see Table 2. Observe that the performance of Boscia on the Optimal Problems decreases when $\log (\text{Tr}(X^{-1}))$ is used as the objective. It should be noted that whether generalized self-concordance holds for the logarithm of the trace inverse (potentially limited to a subset of its domain) is still an open question.

As previously stated, the Co-BnB framework was developed for the case where the number of parameters $n$, and consequently the number of allowed experiment $N$, is significantly smaller than the number of experiments $m$. There are no advanced Branch-and-Bound algorithmic strategies proposed, such as improved traverse strategies or branching decisions. A greater value of $N$ naturally increases the size of the tree and the number of nodes to be processed. Boscia has the advantage here since it finds many integer feasible points while solving the relaxations which have the potential to improve the incumbent. A better incumbent, in turn, lets us prune non-improving nodes early on.

Observe that the curves for the two Outer Approximation approaches also flatten out. In addition, their increase at the start is not as sharp as for the two Branch-and-Bound approaches. Keep in mind that while the relaxations of the two Branch-and-Bound approaches keep the simple feasible region intact, the Outer Approximation methods add many additional constraints, i.e. the cuts. This results in larger MIPs to be solved. Furthermore, these cuts are dense leading to further difficulty for the MIP solvers.

Aside from the performance comparison of the solvers, we investigate how the problems themselves compare to each other and if the difficulty of the instances is solver-dependent or if there are clear trends.
Figure 1 The accumulated number of instances solved to optimality over time for the A-Optimal Problem and D-Optimal Problem with both data sets. The upper plot shows the number of instances solved on the problem with independent data and the bottom the number of solved instances on the correlated data.
Figure 2 The accumulated number of instances solved to optimality over time for the A-Fusion Problem and D-Fusion Problem with both data sets. The upper plot shows the number of instances solved on the problem with independent data and the bottom the number of solved instances on the correlated data.
Table 1 Comparing the performance of Boscia, Pajarito, SCIP and Co-BnB on the different problems and the different data sets, i.e. A-Fusion (AF), D-Fusion (DF), A-Optimal (AO) and D-Optimal (DO). One data set contains independent data, the other has correlated data. The instances for each problem are split into increasingly smaller subsets depending on their minimum solve time, i.e. the minimum time any of the solvers took to solve it. The cut-offs are at 0 seconds (all problems), took at least 10 seconds to solve, 100 s, 1000 s and lastly 2000 s. Note that if none of the solvers terminates on any instance of a subset, the corresponding row is omitted from the table. The average time is taken using the geometric mean shifted by 1 second. Also, note that this is the average time over all instances in that group, i.e. it includes the time outs.

<table>
<thead>
<tr>
<th>Type</th>
<th>Corr.</th>
<th>Solved after (s)</th>
<th># inst.</th>
<th>% solved</th>
<th>Time (s)</th>
<th>% solved</th>
<th>Time (s)</th>
<th>% solved</th>
<th>Time (s)</th>
<th>% solved</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AO</td>
<td>no</td>
<td>0</td>
<td>50</td>
<td>58 %</td>
<td>208.53</td>
<td>42 %</td>
<td>640.68</td>
<td>14 %</td>
<td>1901.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>37</td>
<td>43 %</td>
<td>973.15</td>
<td>22 %</td>
<td>2284.39</td>
<td>0 %</td>
<td>3600.05</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>30</td>
<td>30 %</td>
<td>2084.72</td>
<td>3 %</td>
<td>3531.73</td>
<td>0 %</td>
<td>3600.05</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000</td>
<td>24</td>
<td>12 %</td>
<td>3221.62</td>
<td>0 %</td>
<td>3602.71</td>
<td>0 %</td>
<td>3600.07</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000</td>
<td>22</td>
<td>5 %</td>
<td>3548.1</td>
<td>0 %</td>
<td>3602.94</td>
<td>0 %</td>
<td>3600.07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AO</td>
<td>yes</td>
<td>0</td>
<td>50</td>
<td>82 %</td>
<td>98.5</td>
<td>50 %</td>
<td>541.22</td>
<td>20 %</td>
<td>1591.74</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>37</td>
<td>76 %</td>
<td>330.67</td>
<td>41 %</td>
<td>1145.77</td>
<td>11 %</td>
<td>2777.96</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>23</td>
<td>61 %</td>
<td>1359.18</td>
<td>9 %</td>
<td>3441.78</td>
<td>0 %</td>
<td>3600.05</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000</td>
<td>13</td>
<td>31 %</td>
<td>2922.48</td>
<td>0 %</td>
<td>3600.16</td>
<td>0 %</td>
<td>3600.02</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000</td>
<td>10</td>
<td>10 %</td>
<td>3596.95</td>
<td>0 %</td>
<td>3600.19</td>
<td>0 %</td>
<td>3600.02</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AF</td>
<td>no</td>
<td>0</td>
<td>50</td>
<td>80 %</td>
<td>54.82</td>
<td>78 %</td>
<td>82.96</td>
<td>50 %</td>
<td>1591.74</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>31</td>
<td>68 %</td>
<td>330.67</td>
<td>41 %</td>
<td>1145.77</td>
<td>11 %</td>
<td>2777.96</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>21</td>
<td>52 %</td>
<td>1277.15</td>
<td>48 %</td>
<td>1596.99</td>
<td>0 %</td>
<td>3600.04</td>
<td>0 %</td>
<td>3600.07</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000</td>
<td>13</td>
<td>23 %</td>
<td>3081.0</td>
<td>15 %</td>
<td>3514.57</td>
<td>0 %</td>
<td>3600.05</td>
<td>0 %</td>
<td>3600.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000</td>
<td>10</td>
<td>0 %</td>
<td>3600.13</td>
<td>10 %</td>
<td>3573.82</td>
<td>0 %</td>
<td>3600.06</td>
<td>0 %</td>
<td>3600.12</td>
</tr>
<tr>
<td>AF</td>
<td>yes</td>
<td>0</td>
<td>50</td>
<td>26 %</td>
<td>1359.35</td>
<td>50 %</td>
<td>370.57</td>
<td>20 %</td>
<td>1312.66</td>
<td>14 %</td>
<td>1471.59</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>39</td>
<td>13 %</td>
<td>2684.24</td>
<td>36 %</td>
<td>1421.25</td>
<td>0 %</td>
<td>3600.06</td>
<td>3 %</td>
<td>3375.28</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>35</td>
<td>9 %</td>
<td>3123.03</td>
<td>29 %</td>
<td>2225.95</td>
<td>0 %</td>
<td>3600.07</td>
<td>0 %</td>
<td>3600.03</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000</td>
<td>29</td>
<td>0 %</td>
<td>3600.06</td>
<td>14 %</td>
<td>3393.85</td>
<td>0 %</td>
<td>3600.08</td>
<td>0 %</td>
<td>3600.04</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000</td>
<td>28</td>
<td>0 %</td>
<td>3600.06</td>
<td>11 %</td>
<td>3467.38</td>
<td>0 %</td>
<td>3600.08</td>
<td>0 %</td>
<td>3600.04</td>
</tr>
<tr>
<td>DF</td>
<td>no</td>
<td>0</td>
<td>50</td>
<td>94 %</td>
<td>3.32</td>
<td>86 %</td>
<td>38.28</td>
<td>14 %</td>
<td>1576.5</td>
<td>50 %</td>
<td>333.25</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>6</td>
<td>50 %</td>
<td>858.17</td>
<td>17 %</td>
<td>3336.0</td>
<td>0 %</td>
<td>3613.96</td>
<td>0 %</td>
<td>3600.21</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>5</td>
<td>40 %</td>
<td>1408.69</td>
<td>0 %</td>
<td>3602.04</td>
<td>0 %</td>
<td>3614.84</td>
<td>0 %</td>
<td>3600.23</td>
</tr>
<tr>
<td>DF</td>
<td>yes</td>
<td>0</td>
<td>50</td>
<td>60 %</td>
<td>50.68</td>
<td>54 %</td>
<td>185.07</td>
<td>14 %</td>
<td>1761.18</td>
<td>28 %</td>
<td>753.56</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>24</td>
<td>17 %</td>
<td>2534.26</td>
<td>8 %</td>
<td>2944.52</td>
<td>0 %</td>
<td>3605.93</td>
<td>0 %</td>
<td>3600.05</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>23</td>
<td>13 %</td>
<td>3151.98</td>
<td>4 %</td>
<td>3459.68</td>
<td>0 %</td>
<td>3600.05</td>
<td>0 %</td>
<td>3600.05</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000</td>
<td>22</td>
<td>9 %</td>
<td>3416.05</td>
<td>5 %</td>
<td>3452.39</td>
<td>0 %</td>
<td>3606.45</td>
<td>0 %</td>
<td>3600.05</td>
</tr>
<tr>
<td>D</td>
<td>no</td>
<td>0</td>
<td>50</td>
<td>74 %</td>
<td>81.07</td>
<td>58 %</td>
<td>442.28</td>
<td>24 %</td>
<td>732.57</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>33</td>
<td>61 %</td>
<td>479.16</td>
<td>36 %</td>
<td>2199.09</td>
<td>6 %</td>
<td>1182.22</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>21</td>
<td>43 %</td>
<td>1471.25</td>
<td>10 %</td>
<td>3525.97</td>
<td>5 %</td>
<td>1083.49</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000</td>
<td>6</td>
<td>33 %</td>
<td>2691.04</td>
<td>0 %</td>
<td>3600.88</td>
<td>0 %</td>
<td>2735.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>yes</td>
<td>0</td>
<td>50</td>
<td>100 %</td>
<td>1.26</td>
<td>68 %</td>
<td>223.34</td>
<td>10 %</td>
<td>755.88</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>7</td>
<td>100 %</td>
<td>24.66</td>
<td>0 %</td>
<td>3600.22</td>
<td>0 %</td>
<td>492.01</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

It can be observed in Table 1 that most solvers solve fewer instances under the A-criterion. The notable exception is Pajarito on the A-Optimal Problem where it solves more instances to optimality compared to the D-criterion. It should be noted, however, that Pajarito encountered slow progress for many instances under the A-criterion because too many cuts were added or the cuts were too close, i.e. the normal vectors of the hyperplane were too parallel to each other.

Taking a look at some example contour plots shown in Figure 3, we observe that the contour lines for the A-criterion are steeper than those of the D-criterion for both the Optimal Problem and the Fusion Problem, respectively. This points to the condition number increasing.
Table 2 Comparing the performance of Boscia, Pajarito, SCIP and Co-BnB on A-Optimal and A-Fusion problem with both data sets, i.e. independent and correlated data.

In contrast to Table 1, Boscia and SCIP also have \( \log(\text{Tr}(X^{-1})) \) as objective function.

The instances for each problem are split into increasingly smaller subsets depending on their minimum solve time, i.e. the minimum time any of the solvers took to solve it. The cut-offs are at 0 seconds (all problems), took at least 10 seconds to solve, 100 s, 1000 s and lastly 2000 s. Note that if none of the solvers terminates on any instance of a subset, the corresponding row is omitted from the table. The average time is taken using the geometric mean shifted by 1 second. Also, note that this is the average time over all instances in that group, i.e. it includes the time outs.

<table>
<thead>
<tr>
<th>Type</th>
<th>Corr.</th>
<th>Solved after (s)</th>
<th># inst.</th>
<th>% solved</th>
<th>Time (s)</th>
<th>% solved</th>
<th>Time (s)</th>
<th>% solved</th>
<th>Time (s)</th>
<th>% solved</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AO no</td>
<td>0</td>
<td>50</td>
<td>56 %</td>
<td>325.09</td>
<td>42 %</td>
<td>640.68</td>
<td>14 %</td>
<td>1901.7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>40 %</td>
<td>952.21</td>
<td>28 %</td>
<td>1789.44</td>
<td>0 %</td>
<td>3000.05</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>33 %</td>
<td>2127.44</td>
<td>12 %</td>
<td>2986.04</td>
<td>0 %</td>
<td>3000.06</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000</td>
<td>26</td>
<td>5202.8</td>
<td>0 %</td>
<td>3602.5</td>
<td>0 %</td>
<td>3000.06</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000</td>
<td>24</td>
<td>5472.93</td>
<td>0 %</td>
<td>3602.7</td>
<td>0 %</td>
<td>3000.07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AO yes</td>
<td>0</td>
<td>50</td>
<td>64 %</td>
<td>185.28</td>
<td>50 %</td>
<td>541.22</td>
<td>20 %</td>
<td>1591.74</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>38</td>
<td>613.98</td>
<td>34 %</td>
<td>1669.08</td>
<td>0 %</td>
<td>3000.05</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>29</td>
<td>1441.98</td>
<td>14 %</td>
<td>3346.09</td>
<td>0 %</td>
<td>3000.05</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000</td>
<td>19</td>
<td>5533.58</td>
<td>0 %</td>
<td>3600.11</td>
<td>0 %</td>
<td>3000.04</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AF no</td>
<td>0</td>
<td>50</td>
<td>84 %</td>
<td>31.27</td>
<td>78 %</td>
<td>82.96</td>
<td>12 %</td>
<td>2006.81</td>
<td>38 %</td>
<td>477.72</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>56</td>
<td>270.78</td>
<td>56 %</td>
<td>1074.55</td>
<td>0 %</td>
<td>3000.03</td>
<td>0 %</td>
<td>3000.06</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>43</td>
<td>1457.98</td>
<td>21 %</td>
<td>2474.08</td>
<td>0 %</td>
<td>3000.05</td>
<td>0 %</td>
<td>3000.08</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000</td>
<td>11</td>
<td>3553.67</td>
<td>11 %</td>
<td>3570.97</td>
<td>0 %</td>
<td>3000.06</td>
<td>0 %</td>
<td>3000.12</td>
<td></td>
</tr>
<tr>
<td>AF yes</td>
<td>0</td>
<td>50</td>
<td>52 %</td>
<td>207.34</td>
<td>50 %</td>
<td>370.57</td>
<td>20 %</td>
<td>1132.66</td>
<td>26 %</td>
<td>934.9</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>33</td>
<td>1780.79</td>
<td>24 %</td>
<td>2097.49</td>
<td>0 %</td>
<td>3000.07</td>
<td>3 %</td>
<td>3278.79</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>29</td>
<td>2656.55</td>
<td>21 %</td>
<td>2975.17</td>
<td>0 %</td>
<td>3000.07</td>
<td>0 %</td>
<td>3000.04</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000</td>
<td>26</td>
<td>3461.56</td>
<td>12 %</td>
<td>3457.38</td>
<td>0 %</td>
<td>3000.07</td>
<td>0 %</td>
<td>3000.05</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2000</td>
<td>25</td>
<td>3545.06</td>
<td>8 %</td>
<td>3513.62</td>
<td>0 %</td>
<td>3000.07</td>
<td>0 %</td>
<td>3000.05</td>
<td></td>
</tr>
</tbody>
</table>

In terms of the data, one could assume that all problems would be easier to solve with independent data. Noticeably, this is not the case. Rather, it differs for the two problem types. The Fusion Problems are easier with independent data, the Optimal Problems are more often solved with the correlated data. Figures 4a–4d depict the progress of the incumbent, lower bound, and dual gap within Boscia for selected instances of each combination of problem and data type.

Interestingly, the independent data leads to proof of optimality, i.e. the optimal solution is found early on and the lower bound has to close the gap, regardless of the problem, see Figures 4a and 4c. The difference in problems has, however, an impact on how fast the lower bound can catch up with the incumbent. Comparing Figures 4a and 4c, the lower bound curve in Figure 4c gets closer to the incumbent initially, i.e. in the first 2 seconds, and flattens more compared to the lower bound curve in Figure 4a. The reason for this difference is likely that in the case of the Optimal Problem there is a larger region around the optimal solution where the corresponding points/designs \( x \) provide roughly the same information. These other candidates have to be checked to ensure the optimality of the incumbent and thus the solving process slows down. If we can prove either strong convexity or sharpness for the objectives, this can potentially be used to speed up the progression of the lower bound. Currently, Boscia can only utilize strong convexity but the adaptation for sharpness is in development.

On the other hand, the correlated data leads to solution processes that are very incumbent-driven, i.e. most improvement on the dual gap stems from the improving incumbent, not from the lower bound, as seen in Figures 4b and 4d. Incumbent-driven solution processes
can be identified by the dual gap making sudden jumps and the absence of (a lot of) progress between these jumps. As before, the solution process speed depends on the problem. In Figure 4b, the dual gap makes big jumps throughout most of the solving process, in contrast to the dual gap Figure 4d. For both problems, this indicates that the optimal solution is in the interior. In the case of the Fusion Problem, the lower bound is increasing slower compared to the Optimal Problem in Figure 4b. This points to the existence of many points yielding a similar objective value. The key ingredients for improvement will be the incorporation of more sophisticated primal heuristics and further investigation on how to improve the lower bound progress.

6 Conclusion

We proposed a new approach for the Optimal Experiment Design Problem based on the Boscia framework, and show its performance compared to other MINLP approaches. In addition, it also performs better compared to the approach specifically developed for the OEDP, in particular for large-scale instances and a larger number of parameters. This superiority can be explained by the fact that Boscia keeps the structure of the problem intact and that it utilizes a combinatorial solver to find integer feasible points at each node.

References

As stated in Section 3, we need to prove that the objective function resulting from the A-criterion is generalized self-concordant to establish convergence of Frank-Wolfe. In fact, we can prove the statement for the General-Trace-Inverse (GTI) \( g(x) = \text{Tr}(X^{-p}) \) for \( p > 0 \).

First, we need to compute the first three partial derivatives of \( g \). To help us in this endeavor, we prove the following lemma.

\textbf{Lemma 8.} Let \( r \in \mathbb{R} \) and let the matrix \( X \) be diagonalizable. Then, we can define the function \( f(X) = \text{Tr}(X^r) \). The gradient of \( f \) is then

\[ \nabla f(x) = rX^{r-1} \]
Proof. We use the definition of $X^r$ to prove the result.

$$\text{Tr}(X^r) = \text{Tr} \left( \sum_{k=0}^{\infty} \frac{r^k}{k!} \log(X)^k \right) = \sum_{k=0}^{\infty} \frac{r^k}{k!} \text{Tr}(\log(X)^k)$$

Thus, using the sum rule, $(\text{Tr}(X^n))' = nX^{n-1}$ for any positive integer $n$ and $\nabla \log(X) = \nabla XX^{-1}$, we have

$$\nabla \text{Tr}(X^r) = \sum_{k=1}^{\infty} \frac{r^k}{k!} k \log(X)^{k-1} I X^{-1}$$

$$= r \left( \sum_{k=0}^{\infty} \frac{r^k}{k!} \log(X)^k \right) X^{-1}$$

$$= r X^r X^{-1} = r X^{r-1}$$

This concludes the proof.

Let $A \in S_n^{++}, B \in S_n^+$ and $t \in \mathbb{R}$ such that $A + tB \in S_n^{++}$. We define $h(t) = g(A + tB)$. For the proof of self-concordance, we need the first three derivatives of $h(t)$.

$$h(t) = \text{Tr}((A + tB)^{-p}) = \text{Tr} \left( ((A + tB)^{-1})^p \right)$$

Using Lemma 8 for the first derivative, yields:

$$h'(t) = \text{Tr} \left( p ((A + tB)^{-1})^{p-1} (-1)(A + tB)^{-1}B(A + tB)^{-1} \right)$$

$$= -p \text{Tr} \left( ((A + tB)^{-1})^p B(A + tB)^{-1} \right).$$

For the second derivative, we find

$$h''(t) = -p \text{Tr} \left( p ((A + tB)^{-1})^{p-1} (-1)(A + tB)^{-1}B(A + tB)^{-1}B(A + tB)^{-1} \right.$$ 

$$+ ((A + tB)^{-1})^p B(-1)(A + tB)^{-1}B(A + tB)^{-1} \right)$$

$$= p \text{Tr} \left( (p+1) ((A + tB)^{-1})^p B(A + tB)^{-1}B(A + tB)^{-1} \right).$$

Restricting $h''$ to $t = 0$, yields:

$$h''(t)_{t=0} = p(p+1) \text{Tr} (A^{-p}BA^{-1}BA^{-1}).$$

Lastly, for the third derivative, we have:

$$h'''(t) = -p(p+1)(p+2) \text{Tr} \left( (A + tB)^{-p}B(A + tB)^{-1}B(A + tB)^{-1}B(A + tB)^{-1} \right).$$

And restricted to $t = 0$, yields:

$$h'''(t)_{t=0} = -p(p+1)(p+2) \text{Tr} \left( A^{-p}BA^{-1}BA^{-1}BA^{-1} \right).$$

Proof of Theorem 7. We define $M(t) = V + tU$ where $V \in \text{dom}(g), U \in S_n^+$ and $t \in \mathbb{R}$ such that $V + tU \in S_n^{++}$. Then:

$$h(t) = g(M(t)) = \text{Tr} \left( M(t)^{-p} \right).$$
From the derivative computation above, we have

\[ h''(t)|_{t=0} = p(p + 1) \text{Tr} \left( V^{-p} U V^{-1} U V^{-1} \right) \]
\[ h'''(t)|_{t=0} = -p(p + 1)(p + 2) \text{Tr} \left( V^{-p} U V^{-1} U V^{-1} U V^{-1} \right) . \]

Thus, the condition we want to satisfy is:

\[ p(p + 1)(p + 2) \text{Tr} \left( V^{-p} U V^{-1} U V^{-1} U V^{-1} U V^{-1} \right) \leq M_f p(p + 1) \text{Tr} \left( V^{-p} U V^{-1} U V^{-1} U V^{-1} \right)^{\nu/2} . \]

Note that

\[ \text{Tr} \left( V^{-p} U V^{-1} U V^{-1} U V^{-1} \right) = \text{Tr} \left( V^{-p} U V^{-1} U V^{-1} U V^{-1} \right) \geq 0 \]

and

\[ \text{Tr} \left( V^{-p} U V^{-1} U V^{-1} U V^{-1} U V^{-1} \right) = \text{Tr} \left( V^{-p} U V^{-1} U V^{-1} \right) \geq 0 \]

by positive definiteness of \( V \) and semi-definiteness \( U \). If the trace is equal to 0, the statement holds directly. Otherwise, we divide both sides by the LHS and first try to lower bound the following fraction:

\[ \frac{\text{Tr} \left( V^{-p} U V^{-1} U V^{-1} \right)^{\nu/2}}{\text{Tr} \left( V^{-p} U V^{-1} U V^{-1} U V^{-1} \right)} . \]

We want to lower-bound this fraction. To that end, we define \( H = V^{-1/2} U V^{-1/2} \) and \( L = V^{-p/2} H \), then:

\[ \frac{\text{Tr} \left( V^{-p} U V^{-1} U V^{-1} \right)^{\nu/2}}{\text{Tr} \left( V^{-p} U V^{-1} U V^{-1} U V^{-1} \right)} = \frac{\text{Tr} (LL^\top)^{\nu/2}}{\text{Tr} (LHL^\top)} . \]

Using Cauchy-Schwartz, we find:

\[ \text{Tr} (LHL^\top) = \langle L^\top L, H \rangle \]
\[ \leq \text{Tr} (L^\top L) \sqrt{\text{Tr}(H^2)} \]
\[ \geq \frac{\text{Tr} (L^\top L)^{\nu/2}}{\text{Tr} (LHL^\top)} \geq \frac{\text{Tr} (LL^\top)^{\nu/2}}{\text{Tr} (L^\top L) \sqrt{\text{Tr}(H^2)}} \]
\[ \text{Tr} (H H) = \text{Tr} \left( V^{p/2} V^{-p/2} H H V^{-p/2} V^{p/2} \right) \]
\[ = \text{Tr} \left( V^{p/2} H L L^\top V^{p/2} \right) \]
\[ \leq \text{Tr} (LL^\top) \sqrt{\text{Tr}(V^{p/2})} \]
\[ \geq \frac{\text{Tr} (LL^\top)^{\nu/2}}{\text{Tr} (LL^\top) \sqrt{\text{Tr}(H^2)}} \geq \frac{\text{Tr} (LL^\top)^{\nu/2}}{\text{Tr} (LL^\top) \sqrt{\text{Tr}(V^{p/2})}} \]
\[ = \text{Tr} (LL^\top)^{\nu/2-3/2} \frac{1}{\sqrt{\|V^p\|_F}} \]
\[ \geq \text{Tr} (LL^\top)^{\nu/2-3/2} \frac{1}{\sqrt{\|V^p\|_2 \sqrt{n}}} . \]
By the assumption that $V$ is a feasible point, we have $\|V\|_2 = \lambda_{\max}(V) \leq a$, thus:

$$\frac{\text{Tr}((LL)^{\nu/2})}{\text{Tr}(LL)} \geq \frac{\text{Tr}(LL^{\nu/2-3/2})}{\sqrt{a^{p}n}}.$$  

Then,

$$M_f(p(p+1))^{\nu/2} \cdot \frac{\text{Tr}(V^{-p}UV^{-1}UV^{-1})^{\nu-2}}{p(p+1)(p+2) \text{Tr}(V^{-p}UV^{-1}UV^{-1})} \geq M_f(p(p+1))^{\nu/2} \cdot \frac{\text{Tr}(LL^{\nu/2-3/2})}{\sqrt{a^{p}n}}.$$  

We can select $\nu = 3$ and re-express the GSC condition as

$$1 \leq M_f(p(p+1))^{1/2} \cdot \frac{1}{(p+2)^2 \sqrt{a^{2p}n}}.$$  

Thus, $g$ is $\left(3, \frac{(p+2)^{\nu/2}n}{\sqrt{a^{p+1}}} \right)$-generalized self-concordant on the set $\{X \in \mathbb{S}^n_{++} | 0 < X \preceq aI\}$.

**Corollary 9.** The function $g(X) = \text{Tr}(X^{-1})$ is $\left(3, \frac{3\sqrt{a^{2n}n}}{\sqrt{2}} \right)$-generalized self-concordant on $\text{dom}(g) = \{X \in \mathbb{S}^n_{++} | 0 < X \preceq aI\}$.

To ensure that Theorem 7 is applicable for the objective of $(A\text{-Opt})$, we show that the maximum eigenvalue of the information matrix is bounded.

**Lemma 10.** Let $x \in \mathcal{P}$. Then,

$$\lambda_{\max}(X(x)) \leq m \max_{j \in [m]} u_j \max_{i \in [m]} \|v_i\|^2,$$

where $u$ denotes the upper bounds and $v_i$ are the rows of the experiment matrix $A$.

**Proof.**

$$\lambda_{\max}(X(x)) = \lambda_{\max} \left( \sum_{i=1}^{m} x_i v_i v_i^\top \right).$$

By the Courant Fischer Min-Max Theorem, we have:

$$\lambda_{\max}(X(x)) \leq \sum_{i=1}^{m} \lambda_{\max}(x_i v_i v_i^\top).$$

Using $x \leq u$, yields:

$$\lambda_{\max}(X(x)) \leq m \max_{j \in [m]} u_j \max_{i \in [m]} \|v_i\|^2.$$  

Combining Theorem 7 and Lemma 10, yields:

**Corollary 11.** The function $g(x) = \text{Tr}(X(x)^{-p})$ is self-concordant on the convex feasible region $\mathcal{P}$.
Level sets of the different criteria

(a) D-Optimal Correlated.
(b) A-Optimal Correlated.
(c) D-Fusion Independent.
(d) A-Fusion Independent.

Figure 3 Example contour plots in two dimensions for objectives of both Optimal Problems with correlated data and both Fusion Problems with independent.
Figure 4  Progress of the incumbent and the lower bound on the left and progress of the dual gap for (a) the A-Optimal Problem with independent data, (b) the D-Optimal with correlated data and the A-Fusion Problem with (c) independent and (d) correlated data.
Barcode Selection and Layout Optimization in Spatial Transcriptomics

Frederik L. Jatzkowski
Martin Luther University Halle-Wittenberg, Germany

Antonia Schmidt
Martin Luther University Halle-Wittenberg, Germany

Robert Mank
Martin Luther University Halle-Wittenberg, Germany

Steffen Schüler
Martin Luther University Halle-Wittenberg, Germany

Matthias Müller-Hannemann
Martin Luther University Halle-Wittenberg, Germany

Abstract

An important special case of the quadratic assignment problem arises in the synthesis of DNA microarrays for high-resolution spatial transcriptomics. The task is to select a suitable subset from a set of barcodes, i.e. short DNA strings that serve as unique identifiers, and to assign the selected barcodes to positions on a two-dimensional array in such a way that a position-dependent cost function is minimized. A typical microarray with dimensions of 768 × 1024 requires 786,432 many barcodes to be placed, leading to very challenging large-scale combinatorial optimization problems.

The general quadratic assignment problem is well-known for its hardness, both in theory and in practice. It turns out that this also holds for the special case of the barcode layout problem. We show that the problem is even hard to approximate: It is MaxSNP-hard. An ILP formulation theoretically allows the computation of optimal results, but it is only applicable for tiny instances. Therefore, we have developed layout constructing and improving heuristics with the aim of computing near-optimal solutions for instances of realistic size. These include a sorting-based algorithm, a greedy algorithm, 2-OPT-based local search and a genetic algorithm. To assess the quality of the results, we compare the generated solutions with the expected cost of a random layout and with lower bounds. A combination of the greedy algorithm and 2-OPT local search produces the most promising results in terms of both quality and runtime. Solutions to large-scale instances with arrays of dimension 768 × 1024 show a 37% reduction in cost over a random solution and can be computed in about 3 minutes. Since the universe of suitable barcodes is much larger than the number of barcodes needed, this can be exploited. Experiments with different surpluses of barcodes show that a significant improvement in layout quality can be achieved at the cost of a reasonable increase in runtime. Another interesting finding is that the restriction of the barcode design space by biochemical constraints is actually beneficial for the overall layout cost.

2012 ACM Subject Classification
Applied computing → Operations research; Theory of computation → Mathematical optimization

Keywords and phrases
Spatial Transcriptomics, Array Layout, Optimization, Computational Complexity, GPU Computing, Integer Linear Programming, Metaheuristics

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.17

1 Introduction

We study a challenging large-scale combinatorial optimization problem that arises in the synthesis of microarrays for high-resolution spatial transcriptomics [26, 41], a rapidly evolving molecular profiling method that allows scientists to measure the gene activity in a tissue
sample and map where that activity occurs. At a more abstract level, we consider a special case of the famous quadratic (semi-)assignment problem. The task is to select a suitable subset from a given universe of barcodes and to assign the selected barcodes to the positions (sites) of an array in such a way that a position-dependent cost function is minimized.

**Motivation and background.** DNA barcodes are short strings of fixed length over the alphabet \( \{A,C,G,T\} \). In bioinformatics, they are commonly used as tags in pooled sequencing experiments to enable the identification of reads originating from the same cell. Applications include the study of gene expression at the single-cell level [17], lineage tracing and screening [16], the exploration of developmental trajectories, progression and anti-tumor drug discovery for cancer therapy [2, 40], DNA data storage [1], and high-resolution spatial transcriptomics [26, 41]. The purpose of a barcode is to act as a unique identifier. This means that barcodes must be as easily distinguishable from each other as possible. They must be robust in experimental environments where errors alter the original barcodes by unintentional substitutions, insertions and deletions. Such errors can occur at all stages of the experimental workflow. They can be introduced during barcode synthesis, during the course of the experiment and in the final sequencing [33]. Modern techniques such as photolithographic microarray synthesis generate barcodes on the array in multiple rounds using computer-controlled micromirrors which nowadays replace physical masks. In each round, a subset of all sites (array positions) is exposed to light to activate oligonucleotides for further synthesis. In round \( i \), a particular nucleotide \( s_i \in \{A,C,G,T\} \) is available to be added to the light-exposed sites. Typically, the nucleotide deposition sequence \( S = s_1, s_2, \ldots, s_K \) is assumed to be periodic, i.e. \( S = (ACGT)^k \) for a sufficiently large \( k \) such that all barcodes can be generated [15]. Barcode synthesis faces serious manufacturing challenges due to unintended illumination effects such as scattered or diffracted light which cause a significant amount of error [21]. Experimental analysis has shown that photolithographic microarray synthesis produces barcodes with a comparatively high nucleotide error rate in the range of 10-20% per base [25]. Figure 1 visualizes the principle of the microarray synthesis and how errors depending on adjacent barcodes occur during this process.

This motivates the combinatorial optimization problem studied in this paper. For a given set of barcodes, we try to optimize the layout of the barcodes on the given array by minimizing the dissimilarity of neighboring barcodes. The main goal of this work is to achieve a significant reduction in barcode synthesis errors through a clever combination of barcode selection and layout optimization.

**Barcode set construction.** The construction of large barcode sets is a problem in itself which has been studied intensively, but is not the focus of this paper. The obvious minimum requirement for barcode sets is that all barcodes are unique. However, they should also be designed so that errors can be corrected. The Levenshtein (or edit distance) of two strings of length \( \ell \) is the minimum number of insertions, deletions, and substitutions required to transform one string into the other [23]. A Levenshtein distance of at least \( d \) between any two barcodes provides the worst-case guarantee that up to \( (d - 1)/2 \) errors can be corrected. Since the length of corrupted barcodes can vary, it is even preferable to use a generalized editing metric, the so-called sequence Levenshtein (SL) distance [3].

In addition, for biochemical reasons barcodes should satisfy several additional constraints such as a narrow GC-content range between 40 and 60%, avoidance of homopolymers and repeats of length \( \geq 3 \), i.e. substrings of the type \( (X)^k, (XY)^k, (XYZ)^k, \ldots \) for \( k \geq 3 \), and a distance from special sequences (primers, promoters, flowcell attachments) [6, 7, 38]. The
construction of large barcode sets can be done by rejection sampling. The crucial observation is that if we generate barcode candidates of sufficient length, say of length 30-40, as uniformly random DNA strings, then any two barcodes will have a relatively large Levenshtein distance with high probability [33]. Hence, one can easily generate as many barcode candidates as needed. In the sampling scheme, barcodes that do not fulfill some of the side constraints can be rejected. We will exploit the possibility to choose from large barcode sets to improve the cost of the layout.

**Contribution.** In this paper, we present the following results.

- By giving an \( L \)-reduction from path-TSP on Hamming spaces we show that the barcode layout problem is MaxSNP-hard and thus also APX-hard. Therefore, it does not admit a polynomial-time approximation scheme (PTAS) unless P=NP.
- We study and compare several lower bounds. These include the LP relaxation of an integer linear programming (ILP) formulation and three combinatorial lower bounds, including an adaptation of the Gilmore-Lawler bound for the QAP [9, 22], a simple combinatorial bound proposed by Kahng et al. [15] and a \( b \)-matching relaxation.
- To solve the barcode layout problem in practice, we consider several approaches: different versions of greedy-type algorithms, a 2-OPT-based local search, and a genetic approach. To achieve reasonable running times for solving large instances with these heuristics, most of them had to be implemented on a GPU.
- We present a computational study on a large array with dimensions 768 × 1024. In experiments, we investigate instances with the same number of barcodes as positions to be filled. To assess the quality of the results, we compare the generated solutions with the expected cost of a random layout and with lower bounds. Interestingly, our
GPU implementation of a greedy algorithm produces promising results in terms of both quality and runtime. Solutions to large instances with $768 \times 1024$ many positions show a 37% reduction in cost over a random solution and can be computed in about 94 seconds. Marginal additional improvements can be achieved by performing a local search starting from the greedy solution at moderate additional computational cost.

Other experiments examine the effect of being able to select barcodes from larger sets with varying degrees of redundancy. We observe that the layout costs can be significantly improved, with a nice trade-off between quality improvement and additional runtime.

In a final experiment, we explore the problem under different side constraints that barcodes have to fulfill. Very interestingly, adding extra constraints on the barcode type can be beneficial to the overall cost of the layout.

Related work. The placement of probes arises in a similar way in the design of DNA, protein, and peptide microarrays. It was first discussed under the name border length minimization problem (BLMP) by Hannenhalli et al. [11]. At that time, the standard fabrication technology used a sequence of masks instead of micromirrors, but the basic principle of synthesis in rounds was the same. Each mask exposes a subset of sites to light, namely exactly those where a nucleotide is to be coupled in the current round. Optical effects such as diffraction or reflection of light can cause unwanted illumination and therefore activation of sites adjacent to those that are intentionally exposed to light. To reduce this risk, one seeks a placement of probes minimizing the sum of border lengths in all masks. The NP-hardness of the BLMP has been shown in [32] (but allowing unbounded alphabets for strings as input, while our alphabet is fixed to size four) and for rectangular and square arrays in [20, 21]. Kahng et al. [15] distinguish between synchronous and asynchronous DNA array synthesis. In synchronous synthesis, the $i$-th period ($ACGT$) of the periodic schedule synthesizes a single nucleotide in each probe, whereas asynchronous array synthesis permits any number between 1 and 4 of nucleotides in any given period, allowing shorter synthesis schedules. Finding an optimal deposition sequence (often referred to as embedding) of probes adds another challenging dimension to the BLMP. The combined optimization of placement and embedding has been studied by [5]. For simplicity and to minimize the number of synthesis cycles, in this paper we always use a fixed deposition schedule with a leftmost embedding strategy.

A few approximation results are known for the BLMP. Li et al. [24] show that the special case of a one-dimensional array (i.e. a single row) can be approximated by a constant factor. For asynchronous synthesis, they show that the BLMP is $\sqrt{n} \log^2 n$-approximable when $n$ probes need to be placed. This approximation guarantee has been improved to a factor of $n^{1/4} \log^2 n$ by Popa et al. [32]. Simple combinatorial lower bounds have been proposed by Kahng et al. [15]. The main difference of the classical placement problem to our version is that the given set of probes is usually fixed, whereas the barcode layout problem has the freedom to select from a much larger set of barcodes. The classical BLMP considers the 4-neighborhood of each site as a proxy for measuring the risk of unintended light exposure. The 8-neighborhood of each site includes not only adjacent sites along edges but also at corners. A natural extension is to apply more general distance functions [4], for example using all sites up to a distance of two or three with an appropriate weighting of the distance.

Carvalho and Rahmann [4] recognized the microarray layout problem as a special case of the quadratic assignment problem (QAP). The barcode layout problem studied in this paper can be formulated in a similar way. The QAP is one of the cornerstone problems in combinatorial optimization with a long and rich research history, see for example the surveys [8, 27, 36]. The QAP has a wide range of applications in facility location, scheduling,
transportation, and placement problems in VLSI design. It was originally introduced by Koopmans and Beckmann in 1957 as a mathematical model for allocating a set of \( n \) facilities to \( n \) locations [18]. The QAP is well-known for its theoretical and practical intractability. It includes hard combinatorial optimization problems such as the traveling salesman problem (TSP), graph partitioning and maximum clique as special cases. Exactly solving instances with \( n > 30 \) is typically infeasible in practice, that is, such instances cannot be solved in reasonable computational time. Sahni and Gonzalez [34] showed that the QAP is NP-hard and even finding an \( \epsilon \)-approximate solution is hard. For the general QAP it is even hard to find locally optimal solutions, it is known to be PLS-complete for a Kernighan-Lin type neighborhood and the 2-OPT (or pair exchange neighborhood) [31, 35]. Due to its difficulty, all kinds of meta-heuristics have been applied to QAP [19]. However, their performance depends strongly on the specific application. These meta-heuristics can be combined with a hierarchical refinement algorithm [21].

Overview. The rest of this paper is organized as follows. In Section 2, we formally introduce the barcode layout optimization problem. Then, in Section 3 we study its computational complexity and show its MaxSNP-hardness. Section 4 discusses several lower bounds. As the problem is computationally hard we consider several heuristic approaches in Section 5. In Section 6, we present a computational study evaluating these heuristics on large instances. Finally, in Section 7, we conclude with a brief summary. Source code and data are made freely available at https://github.com/uni-halle/BarLay.

2 Problem Definition

In order to provide a formal problem definition, we first have to formalize the term barcode. For our purposes, a barcode is a word of fixed length \( \ell \) over the alphabet \{A, C, G, T\}. We define a barcode layout as a function that assigns barcodes to array positions.

Definition 1 (barcode layout). Let \( w, h \in \mathbb{N} \) be dimensions of a two-dimensional array and \( B \) with \(|B| \geq w \cdot h \) a set of barcodes of length \( \ell \). The injective layout function \( L(x,y) \) assigns a barcode \( b \in B \) to each position \((x,y)\) with \( 1 \leq x \leq w \) and \( 1 \leq y \leq h \).

During photolithographic synthesis of a microarray, each barcode \( b \) of length \( \ell \) will be uniquely associated with its synthesis schedule \( s(b) \), which is a bitstring of length \( \leq 4\ell \). This schedule determines in which synthesis cycles the barcodes should grow on the microarray. Figure 2 shows an example of barcodes and their associated synthesis schedules. Insertion errors can occur if a barcode is growing in a synthesis cycle it is not supposed to. Typically this happens when a neighboring barcode is illuminated and the light scatters. In this case, the synthesis schedules of the two barcodes are different for the cycle in question. Consequently, the risk of unwanted insertions can be quantified by the Hamming distance of the synthesis schedules. This leads to a natural distance measure between synthesis schedules.
Barcode Layout Optimization

Definition 2 (synthesis distance of barcodes). Let \( b_1 \) and \( b_2 \) be two barcodes. The synthesis distance \( d(b_1, b_2) \) is the Hamming distance of their synthesis schedules \( s(b_1) \) and \( s(b_2) \).

In this paper, we consider the 8-neighborhood of a position to be the set of positions at which unwanted insertions are most likely to occur.

Definition 3 (8-neighborhood). For a position \((x, y)\) on a \(w \times h\)-array, the set

\[
N(x, y) := \{(\tilde{x}, \tilde{y}): |x - \tilde{x}| \leq 1 \land |y - \tilde{y}| \leq 1\} \cap \{(x, y)\} \cap \{1, \ldots, w\} \times \{1, \ldots, h\}
\]

contains the 8-neighborhood around \((x, y)\).

Now, we define a local cost function for each barcode position.

Definition 4 (local cost of a neighborhood). Let \( L \) be a barcode layout on a \(w \times h\)-array. We define the local cost at position \((x, y)\) as

\[
\text{cost}(L, x, y) = \sum_{(\tilde{x}, \tilde{y}) \in N(x, y)} d(L(x, y), L(\tilde{x}, \tilde{y})).
\]

Finally, we are able to define the \textsc{Barcode-Layout} minimization problem:

Definition 5 (\textsc{Barcode-Layout} problem). Input: Set \( B \) of barcodes of length \( \ell \). Array dimensions \( w, h \in \mathbb{N} \) where \(|B| \geq wh\). Output: Barcode layout \( L \) that minimizes

\[
\text{cost}(L) = \sum_{x=1}^{w} \sum_{y=1}^{h} \text{cost}(L, x, y).
\]

3 Computational Complexity

Before presenting our algorithmic approaches, we would like to convey our theoretical findings on the complexity of the \textsc{Barcode-Layout} problem. We show that every problem in MaxSNP is L-reducible to the \textsc{Barcode-Layout} problem and thus, that the \textsc{Barcode-Layout} problem is MaxSNP-hard [29].

Definition 6 (L-reduction [29, 37]). Consider two optimization problems \( A \) and \( B \). \( A \) is said to be L-reducible to \( B \) if there exist two positive constants \( \alpha, \beta \) and two polynomial-time functions \( f, g \) such that
1. for each input \( x \) to \( A \), \( f(x) \) is a valid input to \( B \)
2. \( \text{opt}_B(f(x)) \leq \alpha \cdot \text{opt}_A(x) \)
3. if \( y \) is a solution to \( B \) on the input \( f(x) \), then \( g(x, y) \) must be a valid solution to \( A \)
4. \( |\text{opt}_A(x) - \text{cost}_A(x, g(x, y))| \leq \beta \cdot |\text{opt}_B(f(x)) - \text{cost}_B(f(x), y)| \).

For our proof, we are going to perform an L-reduction from the \textsc{path-TSP} on Hamming spaces to our \textsc{Barcode-Layout} problem.

Definition 7 (\textsc{path-TSP} on Hamming spaces). Given \( n \) cities \( i = 1, 2, \ldots, n \) with coordinates \( c_i \in \{0, 1\}^k \) and pairwise distances \( d_{ij} = d_{\text{Hamming}}(c_i, c_j) \), find a Hamiltonian path of minimum total length.
Trevisan [37] gave a proof for the MaxSNP-hardness of the related TSP on Hamming spaces, in which one looks for a Hamiltonian cycle instead of a Hamiltonian path. He described an L-reduction from (1,2)-TSP to TSP on Hamming spaces. It is straight-forward to apply the same technique to L-reduce (1,2)-path-TSP to path-TSP on Hamming spaces. As (1,2)-path-TSP has been shown to be MaxSNP-hard by Papadimitriou and Yannakakis [30] the following lemma holds.

Lemma 8. path-TSP on Hamming spaces is MaxSNP-hard.

Having established this, we can now prove that BARCODE-LAYOUT is MaxSNP-hard by means of an L-reduction of path-TSP on Hamming spaces.

Theorem 9. BARCODE-LAYOUT is MaxSNP-hard.

Proof. Consider an arbitrary instance $x$ to path-TSP on Hamming spaces. We construct an input $f(x)$ to the BARCODE-LAYOUT problem in linear time such that every layout $y$ to $f(x)$ is associated to some Hamiltonian path $g(x,y)$. For this purpose we transform the coordinates $c = c_1, \cdots, c_k$ of each city in $x$ to a barcode of length $2k$ by applying the following substitutions:

1 $\mapsto AC$ with $s(AC) = 1100$
2 $\mapsto AG$ with $s(AG) = 1010$

Consequently, the synthesis distance of two barcodes is twice the Hamming distance of the original coordinates. Furthermore, the objective value $cost(y)$ of each layout is exactly twice the objective value of the corresponding Hamiltonian path $g(x,y)$, as each synthesis distance between neighboring barcodes is included twice in the cost function (see Definition 6). Thus, the transformation is isometric except for a constant factor of four. We now arrange these constructed barcodes on an $n \times 1$-array. Since the height coordinate is collapsed into a single value, the solution of the input to BARCODE-LAYOUT will provide a permutation of the input barcodes. The cost of this permutation is determined by the sum of the synthesis distances between successive barcodes. If we convert the barcodes in this permutation back to the original Hamming coordinates, we obtain an equivalent solution to path-TSP, where the sum of Hamming distances between successive Hamming coordinates is minimized. We still need to prove that this transformation is indeed an L-reduction. The application of the above substitutions represents the function $f$, while the application of the obtained barcode order to the original Hamming coordinates is the function $g$. Furthermore, the conditions 2 and 4 in Definition 6 hold true for $\alpha = \beta = 4$. So we have shown that BARCODE-LAYOUT is MaxSNP-hard, even if one of the dimensions is collapsed to a single value.

4 Lower bounds

As the optimization problem is hard, we are interested in lower bounds for the best possible cost of a layout. We consider four different lower bounds and sketch the underlying ideas. In Section 6, a comparison of these bounds in terms of strength and computational effort will be presented. For ease of exposition, we assume in this section that the number of barcodes equals the number of array positions, i.e. $|B| = wh$.

LP lower bound. As mentioned before, the BARCODE-LAYOUT problem is a special case of the QAP and therefore can also be formulated as an ILP. Here, we use Lawler’s linearization [22] which requires $O(wh|B|^2)$ variables. While solving these ILPs is only feasible for very small instances with state of the art solvers like Gurobi, we hope that at least the LP relaxation can be computed for larger instances.
A quadratic formulation of the BARCODE-LAYOUT minimization problem uses assignment variables $x_{ijb}$ where $(i, j)$ is a position on the array and $b \in B$ a barcode. The variable $x_{ijb}$ shall be 1 if and only if barcode $b$ is assigned to position $(i, j)$. In a standard formulation of the QAP, quadratic terms such as $x_{ijb}x_{i'j'b'}$ are used to determine whether the barcodes $b$ and $b'$ are adjacent in the positions $(i, j)$ and $(i', j')$. To linearize these quadratic terms we introduce 0/1-variables $y_{(ijb)(i'j'bx)}$ such that $y_{(ijb)(i'j'bx)} = x_{ijbx_{i'j'b'x}}$.

Based on these variables, we obtain the following formulation:

$$
\min \quad \sum_{i=1}^{w} \sum_{j=1}^{h} \sum_{b \in B} \sum_{b' \in B, (i', j') \in N(i,j)} d(b, b') y_{(ijb)(i'j'bx)} \quad \quad (1)
$$

s.t. $$
\sum_{b \in B} x_{ijb} = 1 \quad \quad \forall i \in \{1,\ldots,w\}, \forall j \in \{1,\ldots,h\} \quad (2)
$$

$$
\sum_{i=1}^{w} \sum_{j=1}^{h} x_{ijb} = 1 \quad \quad \forall b \in B \quad (3)
$$

$$
\sum_{i=1}^{w} \sum_{j=1}^{h} \sum_{b \in B} \sum_{b' \in B, (i', j') \in N(i,j),} x_{ijb} + x_{i'j'b'} - y_{(ijb)(i'j'bx)} \leq 1 \quad \quad \forall \in \{1,\ldots,w\}, \forall j \in \{1,\ldots,h\}, \forall b \neq b' \in B, (i', j') \in N(i,j) \quad (4)
$$

$$
\sum_{i=1}^{w} \sum_{j=1}^{h} \sum_{b \in B} \sum_{b' \in B, (i', j') \in N(i,j),} x_{ijb} + x_{i'j'b'} - 2 \cdot y_{(ijb)(i'j'bx)} \geq 0 \quad \quad \forall \in \{1,\ldots,w\}, \forall j \in \{1,\ldots,h\}, \forall b \neq b' \in B, (i', j') \in N(i,j) \quad (5)
$$

$$
\sum_{i=1}^{w} \sum_{j=1}^{h} \sum_{b \in B} \sum_{b' \in B, (i', j') \in N(i,j),} x_{ijb} \in \{0, 1\}, y_{(ijb)(i'j'bx)} \in \{0, 1\} \quad \quad \forall \in \{1,\ldots,w\}, \forall j \in \{1,\ldots,h\}, \forall b \neq b' \in B, (i', j') \in N(i,j) \quad (7)
$$

Equation 2 ensures that every position $(i, j)$ of the array is assigned with exactly one barcode and Equation 3 ensures that each barcode is assigned to exactly one position. Equation 4 says that exactly $2m$ of the $y_{(ijb)(i'j'bx)}$ variables are set to 1, where $m$ is defined as the number of neighborhood relations (vertical, horizontal and diagonal) in an array of size $w \times h$:

$$
m := 2 \cdot (w - 1) \cdot (h - 1) + w \cdot (h - 1) + (w - 1) \cdot h \quad (8)
$$

Finally, Equations 5 and 6 couple the $x$ and $y$ variables. Equation 5 models the implication $(x_{ijb} = 1 \land x_{i'j'bx} = 1) \Rightarrow y_{(ijb)(i'j'bx)} = 1$ and Equation 6 models the implication $y_{(ijb)(i'j'bx)} = 1 \Rightarrow (x_{ijb} = 1 \land x_{i'j'bx} = 1)$. Consequently, $y_{(ijb)(i'j'bx)} = x_{ijb}x_{i'j'bx}$ as desired.

**Gilmore-Lawler bound.** In the field of QAPs, a classical lower bound is the so-called Gilmore-Lawler bound (GLB) [9, 22]. We adapt it to the special case of the BARCODE-LAYOUT problem. In a first step, we compute for each barcode the neighborhood cost for the best neighbors it can possibly have. The values differ depending on the position of the barcode on the array: in a corner, the barcode chooses 3 neighbors; in a border position 5 neighbors and in the middle 8 neighbors. That gives three coefficients for each barcode $b$: $l_b^{(3)}$ as the optimal cost with 3 neighbors, $l_b^{(5)}$ with 5 and $l_b^{(8)}$ with 8 neighbors. We then have to decide for each barcode whether it is best placed in a corner, at the border or in the middle.

We formulate this problem as an ILP. The binary variables $x_{ijb}^{(3)}, x_{ijb}^{(5)}$ and $x_{ijb}^{(8)}$ indicate for barcode $b$ whether it is best placed in a corner, at the border or in the middle. We then solve a linear assignment problem to choose the optimal position for each barcode with respect to the coefficients $l_b^{(3)}, l_b^{(5)}$ and $l_b^{(8)}$. The following ILP shows how we compute the Gilmore-Lawler bound:
\[
\min \sum_{b \in B} (l_b^{(3)} \cdot x_b^{(3)} + l_b^{(5)} \cdot x_b^{(5)} + l_b^{(8)} \cdot x_b^{(8)})
\]
\[
s. t.
\sum_{b \in B} x_b^{(3)} = 4
\]
\[
\sum_{b \in B} x_b^{(5)} = 2 \cdot (w - 2) + 2 \cdot (h - 2)
\]
\[
\sum_{b \in B} x_b^{(8)} = (w - 2) \cdot (h - 2)
\]
\[
x_b^{(3)} + x_b^{(5)} + x_b^{(8)} = 1 \quad \forall b \in B
\]
\[
x_b^{(3)}, x_b^{(5)}, x_b^{(8)} \in \{0, 1\} \quad \forall b \in B
\]

We take into account how many barcodes with 3 (Equation 10), 5 (Equation 11) or 8 (Equation 12) neighbors we need for a valid layout. We also ensure that each barcode is assigned to exactly one position (Equation 13).

**Kahng bound.** A simple combinatorial idea is to select for each barcode the 8 closest barcodes in the set with respect to the synthesis distance. Since the border and corner positions have fewer neighbors, we have to discard a certain number of values. With \(m\) defined as in Equation 8, we then keep only the \(2m\) values with the smallest weights. It is clear that the total weight cannot exceed the cost of an optimal layout and therefore is a lower bound. In the context of microarray design, this bound has been described by Kahng et al. [15] for a 4-neighborhood. It will be referred to as the *Kahng* bound in the following.

**b-matching bound.** Inspired by the 2-matching relaxation of the TSP, we developed a fourth bound. Let \(H\) be an undirected complete graph on the barcode set with the synthesis distance as edge weights. A subset of the edges is chosen such that each node has a degree of at least 3 and at most 8. The number of edges chosen is equal to the number of neighborhood relations on the array, i.e. \(m\) as in Equation 8. Instead of letting each barcode freely choose its eight closest neighbors as edges, we now ensure that neighboring positions must choose their edges consistently. The resulting bound is twice the sum of the edge weights chosen.

The following ILP shows how we compute the \(b\)-matching bound.

\[
\min 2 \cdot \sum_{b \in B} \sum_{b' \in B, b' \neq b} d(b, b') \cdot x_{bb'}
\]
\[
s. t.
\sum_{b' \in B} \sum_{b' \neq b} x_{bb'} = m
\]
\[
\sum_{b' \in B, b' \neq b} x_{bb'} \leq 8 \quad \forall b \in B
\]
\[
\sum_{b' \in B, b' \neq b} x_{bb'} \geq 3 \quad \forall b \in B
\]
\[
x_{bb'} \in \{0, 1\} \quad \forall b \neq b' \in B
\]

The variables \(x_{bb'}\) indicate neighborhood relations: \(x_{bb'} = 1\) if and only if barcodes \(b\) and \(b'\) are connected by an edge. The number of chosen edges is restricted by Equation 16 where \(m\) is the number of edges defined in Equation 8. Equations 17 and 18 ensure the node degrees.
5 Heuristic Approaches

Since we have shown that the BARCODE-LAYOUT problem is MaxSNP-hard, we decided to focus on developing heuristic approaches.

**LEXSORT.** We first consider a sorting based algorithm, which sorts the input lexicographically and subsequently fills the layout with barcodes column by column. This ensures that vertically neighboring barcodes have a maximum common prefix. These barcodes will be synthesized identically for the length of their common prefix. The runtime of this heuristic is dominated by sorting the input. This approach is therefore applicable to large input sets. However, it has several shortcomings in terms of layout quality. First, it only considers the common prefix of barcodes, which can lead to many missed opportunities for good neighbors, which are synthesized differently in the first few cycles. Second, it only optimizes vertically, leaving the possibility of highly sub-optimal neighbors in all other directions.

**Greedy Algorithms.** Kahng et al. [15] describe a greedy algorithm (which they call ROW-EPITAXIAL) and show that it gives promising results. We adapted their algorithm to our problem by making minor changes. The algorithm fills the layout column by column (originally row by row). For each position \((x, y)\), the algorithm selects some yet unassigned barcode \(b\) that minimizes the local synthesis distance

\[
d(L, x, y, b) = \sum_{(\tilde{x}, \tilde{y}) \in N(x, y)} d(b, L(\tilde{x}, \tilde{y}))
\]

which describes the (partial) cost of placing the barcode \(b\) at position \((x, y)\). When evaluating the sum, we only consider neighboring positions to which we already assigned some barcode in a previous step. When considering the \(i\)-th position \((x, y)\), the algorithm needs to calculate the local synthesis distance for each of the \(|B| - i + 1\) yet unassigned barcodes \(b\). As a result, the algorithm must perform \(O(wh|B|)\) calculations of the local synthesis distance, which can be unacceptably large in practice. For this reason, the original ROW-EPITAXIAL algorithm suggested by Kahng et al. had to use a limited lookahead in order to produce results within a reasonable time. Their idea was to select the best unassigned barcode \(b\) from the first \(\leq 20,000\) unassigned barcodes in lexicographic order. This reduces the number of local synthesis distance evaluations to \(O(wh)\), since 20,000 is a constant.

In contrast to Kahng et al., we used an unlimited lookahead (i.e. we chose \(b\) from all unassigned barcodes). This was achieved by using GPU parallelization.

**2-OPT Local Search.** We developed another algorithm to improve existing layouts based on the well-known local search principle. Starting with some initial layout, we iteratively determine pairs of barcodes that can be swapped to improve the layout cost. In each iteration, we first determine the current best swap partner for each barcode on the array. Improving swaps are then performed sequentially, starting with the one that gives the best improvement. Elements that have already been repositioned in this iteration, or are in the neighborhood of such an element, are not swapped. When there are no more profitable swaps, a local minimum is reached and the algorithm terminates. It is straightforward to extend this idea to the more general situation where we have more barcodes than positions on the layout.

In the first iteration of the local search, we have to calculate the effect of swapping *each* barcode on the layout with *every other* barcode in \(B\). Thus, the computational effort in this iteration grows like \(O(wh|B|)\). In subsequent iterations we can take advantage of the fact that new improving swaps can only occur for swapped barcodes or in their neighborhood.
Table 1 Types of barcode sets used in our experiments. All barcodes have length 34 and have been sampled uniformly at random by rejection sampling as sketched in Section 1. Note that \( 4 \ell = 136 \) is an upper bound on synthesis cycles required but in practice much shorter schedules can be achieved.

<table>
<thead>
<tr>
<th>Set name</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>random</td>
<td>no additional constraints</td>
</tr>
<tr>
<td>GC</td>
<td>GC content is 40-60%</td>
</tr>
<tr>
<td>maxCycles</td>
<td>number of synthesis cycles is less than 93</td>
</tr>
<tr>
<td>no repeats</td>
<td>no homopolymers and no repeats of length ( \geq 3 )</td>
</tr>
<tr>
<td>constrained</td>
<td>GC + maxCycles + no repeats</td>
</tr>
<tr>
<td>distance</td>
<td>constrained + pairwise SL distance at least 9</td>
</tr>
</tbody>
</table>

Genetic Algorithm. In a further effort to improve the quality of layouts, we have developed a genetic algorithm. This approach models biological evolution by modifying a population of 1024 layouts using a combination of selection, crossover and mutation, hopefully converging towards a local optimum [13, 14]. We use tournament selection [12] with 8 participants to select a subset of the population for reproduction. The selected couples then produce two offspring using a crossover operator that is based on the partially mapped crossover operator [39], but favors barcode placements which already have a low local cost function within their neighborhood. The idea of using local cost information for an optimized crossover operator comes from [28] and [10]. Finally, a simple swap operator is applied to these offspring to improve genetic diversity.

6 Experimental Results

In this section, we evaluate the solution quality of our algorithms. For this purpose, we experiment with multiple barcode sets that differ by several additional constraints (see Table 1).

6.1 Comparison of Lower Bounds

We start by comparing the quality of the lower bounds described in Section 4.

Experimental setup. We sampled several subsets from size 10 \( \times \) 10 to size 768 \( \times \) 1024 from the barcode set distance and evaluated the four lower bounds on each subset. To solve the (I)LPs, we used Gurobi in version 9.52.

Results and discussion. Table 2 shows the results of the experiment. We see that the bounds calculated with the (I)LP solver can only be computed for small instances within reasonable time. The LP bound turns out to be much weaker than the other bounds. In contrast, the Kahng and GLB bounds are much stronger and can be evaluated for large instances within reasonable time. The \( b \)-matching is only marginally better than the Kahng bound but scales less well. Thus, we decided to use the Kahng bound in the following experiments.

6.2 Expected Cost of Random Layouts

To get another baseline for the heuristics, we calculated the expected cost of a random layout, where a random barcode is selected from \( B \) and assigned to each array position.
Table 2 Lower bounds for different array sizes and barcode sets of size $w \cdot h$ of type distance. Entries with NA stand for instances which we could not solve within reasonable time and memory.

<table>
<thead>
<tr>
<th>method</th>
<th>10 x 10</th>
<th>15 x 15</th>
<th>20 x 20</th>
<th>100 x 100</th>
<th>768 x 1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>LP</td>
<td>13,216</td>
<td>30,120</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>GLB</td>
<td>21,344</td>
<td>48,202</td>
<td>84,988</td>
<td>1,881,900</td>
<td>119,211,464</td>
</tr>
<tr>
<td>Kahng</td>
<td>20,964</td>
<td>47,744</td>
<td>84,376</td>
<td>1,884,112</td>
<td>119,215,966</td>
</tr>
<tr>
<td>$b$-matching</td>
<td>21,032</td>
<td>47,852</td>
<td>84,676</td>
<td>1,884,904</td>
<td>NA</td>
</tr>
</tbody>
</table>

Experimental setup. To calculate the expected layout cost $E(cost)$, we first need to calculate the expected synthesis distance $E(d)$ between two barcodes sampled uniformly at random from $B$. This value depends on the specific barcode set $B$ and can be calculated by summing over all barcode pairs of this set. We then calculate the expected layout cost using the formula $E(cost) = 2m \cdot E(d)$ with $m$ as defined in Equation 8.

Results and discussion. For the given set of type distance with 768 x 1024 barcodes, this resulted in $E(cost) \approx 254,498,050$ for a 768 x 1024 array. This baseline was also empirically confirmed as the mean cost of 10 randomly generated layouts was 254,485,241.6 with a standard deviation of 14,378.1. We observe that the expected cost is about twice as large as the Kahng bound.

6.3 Comparison of Heuristics

Next, we investigated the quality of our heuristics.

Experimental setup. We ran each heuristic ten times on the barcode set of type distance with 768 x 1024 barcodes and the same layout dimension (thus without any excess on barcodes) and determined the average layout cost. Since the genetic algorithm and the 2-OPT local search require initial layouts, we tested these algorithms first with random initial layouts and afterwards with pre-optimized layouts produced by the unlimited Greedy Algorithm. For each heuristic, we calculated the percentage improvement over the expected layout cost (gain) and the gap to Kahng’s lower bound (gap).

Table 3 Gain (%) gives the improvement over the expected cost of a random layout in percent. Gap (%) shows the optimality gap to the Kahng lower bound in percent.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Average Cost</th>
<th>gain (%)</th>
<th>gap (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lower bound (Kahng)</td>
<td>119,215,966</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Random layout (expected value)</td>
<td>254,498,050</td>
<td>—</td>
<td>113.48</td>
</tr>
<tr>
<td>LEXSORT</td>
<td>218,300,868</td>
<td>14.22</td>
<td>83.11</td>
</tr>
<tr>
<td>Greedy (lookahead 20000)</td>
<td>178,342,888</td>
<td>29.90</td>
<td>49.60</td>
</tr>
<tr>
<td>Greedy (unlimited)</td>
<td>159,937,955</td>
<td>37.16</td>
<td>34.16</td>
</tr>
<tr>
<td>2-OPT Local Search (random initial layout)</td>
<td>181,718,241.2</td>
<td>28.60</td>
<td>52.43</td>
</tr>
<tr>
<td>2-OPT Local Search (pre-optimized initial layout)</td>
<td>159,887,640.4</td>
<td>37.18</td>
<td>34.12</td>
</tr>
<tr>
<td>Genetic Algorithm (random initial layouts)</td>
<td>203,261,559</td>
<td>20.13</td>
<td>70.50</td>
</tr>
<tr>
<td>Genetic Algorithm (pre-optimized initial layout)</td>
<td>163,691,770</td>
<td>35.70</td>
<td>37.31</td>
</tr>
</tbody>
</table>
Table 4 Average runtimes for sequential CPU or parallelized GPU implementations of selected algorithms in Table 3. Sequential runtimes for 2-OPT were estimated from the sequential runtime of a single iteration.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>CPU (sequential)</th>
<th>GPU (parallelized)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEXSORT</td>
<td>0.39 sec</td>
<td>-</td>
</tr>
<tr>
<td>Greedy (unlimited)</td>
<td>7.86 hours</td>
<td>94.1 sec</td>
</tr>
<tr>
<td>2-OPT Local Search (random initial layout)</td>
<td>46.4 days</td>
<td>15.6 min</td>
</tr>
<tr>
<td>2-OPT Local Search (pre-optimized initial layout)</td>
<td>3.7 days</td>
<td>89.5 sec</td>
</tr>
</tbody>
</table>

Results and discussion. Table 3 and Table 4 show the results. We highlight some essential observations.

- We observe that the unlimited Greedy Algorithm produces solutions of high quality, which could be improved marginally by a successive 2-OPT local search at the cost of small additional runtime.
- Interestingly, starting the 2-OPT local search with a random initial layout produces layouts that are local optima, but far worse than the layout produced by our greedy algorithm. This suggests that the local optima of the 2-OPT neighborhood relation differ drastically in their quality.
- Looking into the 2-OPT local search in more detail, we observe that it improved a random initial layout by about 28.60% in 204.1 iterations on average (see Figure 3a) and thus performs better than LEXSORT, but much worse than the unlimited Greedy Algorithm. Using a pre-optimized layout generated by the unlimited Greedy Algorithm, a local optimum was quickly reached after only 12.4 iterations on average, as can be seen in Figure 3b.
- Unfortunately, the Genetic Algorithm could not further reduce the cost of layouts produced by the unlimited Greedy Algorithm. Instead, we observed increasing costs within the first generations and a plateau afterwards. This behaviour is probably caused by the mutation operator, which is applied after each generation. This operator swaps randomly selected barcodes in order to maintain sufficient genetic diversity throughout the population. In highly optimized layouts, this tends to destroy good neighborhoods and create more average ones, increasing the overall cost of the layout.
- GPU parallelization greatly accelerates our algorithms compared to a sequential CPU implementation. We estimate the speedup factor of the 2-OPT algorithm to be more than 4000. The Greedy Algorithm was accelerated by a factor of about 300.

6.4 Barcode Sets with Excess

Our experiments suggest that the greedy algorithm provides a significant improvement in layout cost when compared to random layouts. However, a typical weakness for greedy algorithms is that the choices made later in the process get worse as there are fewer options left. To address this issue, we run an additional experiment where we provide more barcodes to choose from than the Greedy Algorithm needs for the layout.

Experimental setup. Starting with 768×1024 barcodes of type random, we iteratively increased the barcode excess by including more barcodes. In doing so, we generated barcode sets that are 1×, 1.5×, 2×, 2.5×, 3×, 3.5×, and 4× as large as the initial set. We ran the unlimited Greedy algorithm 10 times on each set and determined the average layout cost.
Results and discussion. Figure 4 shows the average layout cost of our greedy heuristic for barcode sets with multiple excess factors. As expected, there is a clear decrease in cost as the size increases. Even a small excess factor of 1.5 has an effect: The cost decreases by about 10.71% compared to using the original set of 768×1024 barcodes (without any excess). The cost decreases by up to 43.42% compared to the baseline as the size of the barcode set increases.

We conclude that even a small excess of barcodes can largely improve the layout quality. Runtime increases linearly with the barcode set size: The runtime for the set of 4·w·h barcodes is 386 seconds and thus about 4 times as long as for the set of w·h barcodes with 94 seconds. Our observations show that our implementation scales well and is even suitable for next-generation microarrays with four million positions.
6.5 Distribution of Layout Cost

Next we analyse how much each layout position \((x, y)\) contributes to the total layout cost. To do this, we first normalized the local cost (see Definition 4) of each position \((x, y)\) by the number of its neighbors (which is either 3, 5, or 8). This allows us to fairly compare the contributions of each layout position regardless of their number of neighbors. For the matter of a clear presentation, we show the normalized local costs averaged over each column \(y\). Thus, the average local cost of column \(y\) is defined as

\[
\frac{1}{w} \sum_{x=1}^{w} \frac{\text{cost}(L, x, y)}{|N(x, y)|},
\]

The larger this value, the more the \(y\)-th column contributes to the layout cost.

Experimental setup. We repeated the previous experiment, now calculating the individual average local costs for each column.

Results and discussion. Figure 5 shows the results of this experiment. We observe a rapid increase in cost in the last columns, when the greedy algorithm has almost no freedom of choice anymore. If we use more barcodes than necessary, this effect disappears. Instead, we see an almost linear change in quality with a slope that decreases with larger barcode sets. These observations confirm our previous conclusions.

6.6 Influence of Barcode Set Type

As mentioned earlier, barcode sets are usually constrained for technical and other reasons. We therefore carried out a final experiment to assess whether our Greedy Algorithm behaves differently on constrained barcode sets.

Experimental setup. We used a barcode set of each type mentioned in Table 1 and evaluate the average layout cost of the unlimited Greedy Algorithm for an array of size 400×500. Each barcode set contained 800,000 barcodes. Similar to the previous experiment, we plot the average local cost of each individual column.
**Results and discussion.** Figure 6 shows the results. There are two distinct groups: The first group with the higher costs consists of the random set, the GC set and the maxCycles set. The distance set, the constrained set and the no repeats set form the second group with the lower cost values. We conclude that constraining the GC content and the number of synthesis cycles has little effect on the barcodes, which is why these sets behave almost the same as the random set. The exclusion of repeats greatly reduces the set of valid barcodes. This means that the average synthesis distance between barcodes is significantly reduced, which translates into lower costs for the generated layouts.

We conclude that using certain constraints on the barcode sets doesn’t have a negative impact on the cost of the layout. On the contrary, the cost even improves compared to a random set of barcodes.

**7 Summary**

In this paper we provided a computational study on the BARCODE- LAYOUT problem that arises in the synthesis of DNA microarrays by photolithography. Experiments on instances of typical size demonstrate that the resulting combinatorial optimization problems are very hard to solve. We obtained the best results with a greedy approach followed by 2-OPT local search which improved on a random layout by 37%. However, the optimality gap with respect to the best lower bound is also relatively large at 34%. We suspect that the available lower bounds are quite weak so that the actual solutions are not too far away from the unknown optimum. Due to the long runtimes of the sequential versions of the Greedy and 2-OPT algorithms, it was crucial to implement variants that are accelerated on a GPU.

Our attempts to further improve the greedy solutions using stochastic meta-heuristics has been rather disappointing. Exemplarily we reported non-competitive results with a genetic algorithm. Future work could explore other meta-heuristics in more detail.

One important finding is that we can benefit greatly from selecting barcodes from a large pool of candidates. This can improve solutions by up to 43% compared to random layouts. Additionally, experiments with realistic barcode sets that meet practical side constraints result in even better layouts than those with randomly chosen barcodes.

**References**


Experimental Analysis of LP Scaling Methods Based on Circuit Imbalance Minimization

Jakub Komárek  
Computer Science Institute, Charles University, Prague, Czech Republic

Martin Koutecký  
Computer Science Institute, Charles University, Prague, Czech Republic

Abstract

Linear programming (LP) is a fundamental problem with rich theory and wide applications. A ubiquitous technique in LP is scaling, where the input instance is transformed in some way to make its solution easier. Dadush et al. [STOC ’20] have recently devised an algorithm which scales the columns of the constraint matrix of a linear program in a way that aims to minimize the circuit imbalance measure, a matrix condition number of growing theoretical interest. They show that this rescaling achieves favorable theoretical guarantees for certain LP algorithms.

We follow up on their work in an experimental manner. First, we have implemented their algorithm, overcoming several engineering obstacles. Next, we have used our implementation to obtain a rescaling of 142 publicly available instances. Finally, we have performed experiments evaluating the effects of the obtained rescalings on the runtime of real-world LP solvers, and we have evaluated their quality with regard to the circuit imbalance measure.

2012 ACM Subject Classification  Theory of computation → Linear programming

Keywords and phrases  Linear programming, scaling, circuit imbalance measure

Digital Object Identifier  10.4230/LIPIcs.SEA.2024.18


Supplementary Material  Software (Source code and experiment data): https://github.com/Jakoma02/lp-scaling-analysis, archived at swh:1:dir:0a90ab07fada3a81f5b7cf470f48db5a951dc855

Funding  Partially supported by project 22-22997S of GA ČR. Computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic. Martin Koutecký: Supported by Charles Univ. project UNCE 24/SCI/008.

1 Introduction

Linear programming (LP) is a core topic in theoretical computer science and optimization with many applications in practice. One of the possible approaches to improve LP algorithms is the use of preconditioning [7] or scaling [19, Section 7.2]. Dadush et al. [9] have shown that the complexity of the LP interior-point method of Vavasis and Ye [23] can be bounded in terms of a certain condition number \( \kappa \) of the constraint matrix \( A \). This condition number can be changed by scaling individual columns of \( A \) by positive real values, and they present an algorithm which finds a positive diagonal matrix \( D \) such that the \( \kappa \) of the rescaled matrix approximates the minimum achievable value.

We follow up on their work by exploring its applicability in practice and describing the effects of the condition number on the performance of practical LP solvers, including ones not based on interior-point methods. The theoretical results of Dadush et al. [9] only directly apply to the particular interior-point method of Vavasis and Ye [23], so why do we test whether it speeds up other methods? We acknowledge this apparent “irrelevance”;
our motivation is simply that the rescaling of Dadush et al. [9] is a significant theoretical
contribution, and it is not a priori obvious whether and how it would affect the performance
of real-world solvers.

1.1 The Circuit Imbalance Measure

The matrix condition numbers $\kappa$ and $\kappa^*$ as defined by Dadush et al. [9] are center to our
work. Consider an LP $\min \{c^T x \mid Ax = b, x \geq 0\}$. A circuit of the vector matroid of the
columns of $A$ is an inclusion-wise minimal set of indices of linearly dependent columns. For
every circuit $C$ of the vector matroid of $A$, we can consider its elementary vector, which is
an element of the kernel of $A$ that has $C$ as its support. The circuit imbalance measure $\kappa$ of
matrix $A$ is the maximum ratio of absolute values of nonzero elements over all elementary
vectors of the vector matroid of $A$.

Scaling the columns of $A$ does not change the set of circuits $\mathcal{C}$. However, the elementary
vectors do change and so $\kappa$ can also change. Viewing $\kappa$ as a measure of how ill-conditioned
the matrix $A$ is, we ask for the best value of $\kappa$ obtainable by choosing a suitable column
scaling. This optimal value is denoted $\kappa^*$.

The interest in circuits and their imbalances has been steadily growing. Since circuits
are a generalization of the edge directions of a polytope, bounds on the lengths of circuit
walks have been of much interest [3, 4]. The computational complexity of finding good
circuit steps has also been considered [18, 2, 5]. In the wake of the paper of Dadush et
al. [9], circuit imbalances have been used to devise a fast first-order LP algorithm [8] or prove
stronger bounds on circuit walks [10]. For more, we refer the reader to the excellent survey
of Ekbatani et al. [11].

1.2 Our Contribution

Our first contribution is an implementation of the approximation rescaling algorithm of
Dadush et al. [9]. We have discovered several non-obvious challenges, such as how exactly
sparsity or floating-point arithmetic comes into play. We discuss the possible approaches
and their trade-offs, paving the way for any possible further engineering effort. Secondly,
we used the implementation to obtain an approximate rescaling of many problems in the
benchmark sets Netlib [13] and MIPLIB [14], considering continuous relaxations in the latter
case. Because of the high (even if polynomial) complexity of the algorithm and size of
the mentioned problems there remain problem instances for which the computation of the
rescaling using our implementation is infeasible in realistic timeframes. Third, we used the
LP solvers GLPK1, Gurobi [15], and SCIP [1] to evaluate the impact of the computed
rescalings on the efficiency of the solvers. We have used all available solving methods and
using both exact and floating-point arithmetic where available. Moreover, we discuss how
successful the algorithm is in reducing the circuit imbalance measure $\kappa$.

Our main finding can be seen in Figure 1. On the one hand, the rescaling did not
consistently speed-up any one solving method. On the other hand, for a great majority (79%)
of instances, the rescaling sped-up at least one of the solving methods, showing possible
promise for practical applicability. Moreover, the greatest achieved speed-up was by a factor
of $13 \times$. It also needs to be said that the compared solving times are sometimes very small
(see the full data for details), possibly decreasing our confidence in meaningfulness of the
measured time change.

1 Open source software available at http://www.gnu.org/software/glpk/glpk.html
Figure 1 The results of the experiment. The rescaling improved the performance of at least one method for 112 out of the total 142 (79%) tested instances, and the best achieved speed-up was by a factor of $13 \times$. However, the rescaling did not consistently help any particular method.

Because no single method consistently benefited from the rescaling, this may be regarded as a negative result. However, we still believe that positive results even for a minority of instances can point to potential applicability of this scaling approach. Conceivably, future research could identify features of instances for which this type of scaling helps, and if these instances could be recognized quickly, solving time improvements could be reaped. A major obstacle in this direction is the time necessary to compute the rescaling itself, which for all considered instances was vastly longer than the time it takes to solve the instance in the first place. However, our goal here was purely exploratory.

Paper Organization

This paper is based on the thesis [17], which contains many more details and helpful illustrations and examples.

2 Preliminaries

2.0.1 Notation

$\mathbb{R}$ denotes the set of real numbers, and $\mathbb{N}$ the set of natural numbers, including zero, and $[n]$ denotes the set $\{1, 2, \ldots, n\}$. Matrices are written as $A \in \mathbb{R}^{m \times n}$ and vectors as $x \in \mathbb{R}^n$. 
All vectors are regarded as column vectors, the notation $\mathbf{x}^\top$ is used to denote row vectors when needed. Vector elements are provided with indices and not written in bold font, as in $\mathbf{x} = (x_1, x_2, \ldots, x_n)^\top$. For $B \subseteq [n]$, let $A_B$ denote the submatrix of $A$ obtained by selecting exactly columns with indices in $B$ and similarly let $\mathbf{v}_B$ denote the subvector of $\mathbf{v}$ obtained by selecting exactly the entries of $\mathbf{v}$ with indices in $B$. Whenever used with the index in bold font, $\mathbf{x}_i$ does not denote the $i$-th coordinate, but it is a vector itself. The notation $\text{Diag}(\mathbf{d})$ means a diagonal matrix with elements of $\mathbf{d}$ on the diagonal. When using the $\leq$ and similar relations ($\geq$, $<$, $>$, $=$) on vectors, we mean that the relation holds coordinate-wise, i.e. $\mathbf{x} \leq \mathbf{y}$ means that $x_i \leq y_i$ for every coordinate $i$. Sets are denoted with capital letters. We use the union symbol strictly for the union of sets, e.g., $A \cup B$. For the set difference of $A$ and $B$, we write $A \setminus B$. If $\mathbb{F}$ is arbitrary field, then the coordinate space whose elements are $n$-tuples of elements of $\mathbb{F}$ is denoted $\mathbb{F}^n$.

2.0.2 Linear Programming

For $A \in \mathbb{R}^{m \times n}$, $\mathbf{b} \in \mathbb{R}^m$ and $\mathbf{c} \in \mathbb{R}^n$, linear programming (LP) in standard form is the problem $\min \{ \mathbf{c}^\top \mathbf{x} \mid A\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}, \mathbf{x} \in \mathbb{R}^n \}$. The standard form is convenient in theoretical contexts, however, in real-world applications, the following modelling form is common. Let $A_{ub} \in \mathbb{R}^{n_1 \times n}$, $A_{eq} \in \mathbb{R}^{m \times n}$, $\mathbf{b}_{ub} \in \mathbb{R}^{m_1}$, $\mathbf{b}_{eq} \in \mathbb{R}^{m_2}$, $\mathbf{c} \in \mathbb{R}^n$ and $\mathbf{l}, \mathbf{u} \in (\mathbb{R} \cup \{-\infty, \infty\})^n$ such that $\mathbf{l} \leq \mathbf{u}$. Then, a linear program in modelling form is

$$\min \{ \mathbf{c}^\top \mathbf{x} \mid A_{ub}\mathbf{x} \leq \mathbf{b}_{ub}, A_{eq}\mathbf{x} = \mathbf{b}_{eq}, \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}, \mathbf{x} \in \mathbb{R}^n \}$$

It is not difficult to see that every LP in standard form is already in modelling form. What is potentially less clear is that also every LP in modelling form can be efficiently converted to an equivalent LP in standard form (using standard techniques, see, e.g., Gärtner and Matoušek [12, Section 4.1]), rendering the two forms easily interchangeable.

2.0.3 Solving Scaled Instances

The motivation for scaling comes from the ability to relate the optima of the original and the scaled instance, as shown here:

**Proposition 1.** Let $A \in \mathbb{R}^{m \times n}$, $\mathbf{b} \in \mathbb{R}^m$ and $\mathbf{c} \in \mathbb{R}^n$ and let $D \in \mathbb{R}^{n \times n}$ be a diagonal matrix with positive entries on the diagonal. Let $P$ be the standard-form LP $\min \{ \mathbf{c}^\top \mathbf{x} \mid A\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0} \}$ and $Q$ the standard-form LP $\min \{ \mathbf{c}^\top D\mathbf{x} \mid (AD)\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0} \}$. Then $\mathbf{x}'$ is an optimum of $P$ if and only if $D^{-1}\mathbf{x}'$ is an optimum of $Q$.

**Proof.** Let $\mathbf{x}'$ be an optimum of $P$. Because $D$ has positive diagonal entries, $D^{-1}$ also has positive diagonal entries, so $D^{-1}\mathbf{x}' \geq \mathbf{0}$ holds. The equality constraint is also satisfied, which follows from $(AD)(D^{-1}\mathbf{x}') = A(DD^{-1})\mathbf{x'} = A\mathbf{x'} = \mathbf{b}$. This proves that $D^{-1}\mathbf{x}'$ is feasible for $Q$. What is left to show is the optimality of $D^{-1}\mathbf{x}'$ for $Q$. For contradiction consider that $Q$ has a better solution $\mathbf{x}''$ such that $\mathbf{c}^\top D\mathbf{x}'' < \mathbf{c}^\top D(\mathbf{D}^{-1}\mathbf{x}')$. Then, $\mathbf{x}''$ is feasible for $P$ and it holds that $\mathbf{c}^\top D\mathbf{x}'' < \mathbf{c}^\top \mathbf{x}'$, contradicting that $\mathbf{x}'$ is an optimum of $P$. The proof of the opposite direction is symmetric. ▲

2.1 Circuit Imbalance Measures

We refer to the thesis [17, Section 2.3] or the book of Oxley [20] for the basics of matroid theory. In Section 1.1, we have already defined the notions of a circuit, its elementary vector, and the circuit imbalance measure $\kappa$. For a circuit $C$, let $\mathbf{g}^C$ denote its elementary vector,
and for a matroid $M$, let $\mathcal{C}(M)$ denote the set of its circuits. The \textit{pairwise circuit imbalance measure} for coordinate pair $i, j \in [n], i \neq j$, is the value

$$
\kappa_{ij}^A(C) := \frac{|g^C_i|}{|g^C_j|} \text{ or 0 if } g^C_i = 0, \quad \kappa_{ij}^A := \max\{\kappa_{ij}^A(C) \mid C \in \mathcal{C}(A), i, j \in C\}.
$$

By convention we set $\kappa_{ij} = 0$ if there is no circuit supporting $i$ and $j$.

Define the \textit{circuit imbalance measure} to be $\kappa_A := \max\{\kappa_{ij}^A \mid i, j \in [n]\}$, and let $\kappa_A(C) := \max\{\kappa_{ij}^A(C) \mid i, j \in C\}$, which satisfies $\kappa_A = \max\{\kappa_A(C) \mid C \in \mathcal{C}(M(A))\}$.

Let $D$ be the set of $n \times n$ real diagonal matrices with positive entries on the diagonal. The \textit{optimal circuit imbalance measure} for $A$ is the number $\kappa_A^* := \inf\{\kappa_A(D) \mid D \in D\}$. If clear from context, we will omit the $A$ subscript or superscript from $\kappa$ and its variants. It is not a priori clear that the infimum is indeed attained, but we will later see that it in fact is.

### 2.2 Algorithm for Optimizing the Circuit Imbalance Measure

Here we give an overview of the scaling algorithm of Dadush et al. [9] that is implemented and used in our experiments. The purpose of the algorithm is to find a diagonal matrix $D$ such that $\kappa_{AD}$ is not much larger that $\kappa_A^*$.

First, it is not difficult to see that we can assume the vector matroid of $A$ is connected, otherwise we can process its components separately.

Next, it follows from Tuncel [22] that approximating $\kappa$ up to a factor of $O^{O(n)}$ is NP-hard. The algorithm provided in the following text finds a $\tilde{\kappa}_A$ satisfying

$$
\tilde{\kappa}_A \leq \kappa_A \leq n(\kappa_A^*)^2 \tilde{\kappa}_A
$$

in polynomial time. Note that this is not a contradiction to the NP-hardness claim, because the approximation ratio depends on $\kappa_A^*$.

The basic idea of the algorithm is as follows. One first shows that if we had the values $\kappa_{ij}^A$, we could efficiently compute the value of $\kappa$. The values $\kappa_{ij}$ are of course still NP-hard to compute, but it is possible to efficiently find certain proxies and use these instead to obtain a good enough approximation $\tilde{\kappa}$. The following theorem states what kind of proxies we are looking for, and what approximation do they yield; the theorem is stated in [9] differently due to the different focus of their work compared to ours, but its substance is the same:

**Theorem 2** ([9, Theorem 2.5]). For every $i, j \in [n]$ let $\tilde{\kappa}_{ij}$ satisfy

$$
\tilde{\kappa}_{ij} \leq \kappa_{ij} \leq (\kappa^*)^2 \tilde{\kappa}_{ij},
$$

Define $\tilde{\kappa} := \max\{\tilde{\kappa}_{ij} \mid i, j \in [n]\}$. Then it holds that $\tilde{\kappa} \leq \kappa \leq n(\kappa^*)^2 \tilde{\kappa}$.

From now on, we will use $\tilde{\kappa}_{ij}$ for any value satisfying (1). While possibly many values satisfy this, the same value is always meant when $\tilde{\kappa}_{ij}$ occurs multiple times. Now we are left with the task of finding the pairwise approximations $\tilde{\kappa}_{ij}$. The solution is actually simple: find any circuit that contains both $i$ and $j$ and estimate $\kappa_{ij}$ based on this circuit:

**Proposition 3** ([9, Corollary 2.13]). Let $i, j \in [n]$ and let $C$ be any circuit such that $i, j \in C$. Then $\kappa_{ij}(C) \leq \kappa_{ij} \leq (\kappa^*)^2 \kappa_{ij}(C)$.

The task of finding such circuit will be the central topic of Section 2.2.1. Computing the diagonal matrix which witnesses an approximation $\tilde{\kappa}^*$ of $\kappa^*$ is more involved, but can still be done efficiently [9, Theorem 2.5]; let us reformulate their result in a more algorithmic way:
Definition 4 (Circuit ratio digraph). The circuit ratio digraph is a directed graph \( \mathcal{G} \hat{G} := ([n], E) \), where \( E := \{(i, j) \in [n]^2 \mid i \neq j, \kappa_{ij} > 0\} \).

Remark 5. From Proposition 3 it follows that \( \kappa_{ij} > 0 \) if and only if \( \hat{\kappa}_{ij} > 0 \). The edge set above can therefore be equivalently defined as \( E := \{(i, j) \in [n]^2 \mid i \neq j, \hat{\kappa}_{ij} > 0\} \).

Theorem 6 ([9, Theorem 2.5]). Consider the optimization program

\[
\begin{aligned}
\min & \hat{t} \\
\text{s.t.} & \hat{\kappa}_{ij} d_{ij} / d_i \leq t & \forall (i, j) \in E(\mathcal{G} \hat{G}) \\
& d > 0.
\end{aligned}
\]

Let \( \hat{d} \) be the optimal value of \( d \) for (2) and \( D := \text{Diag}(\hat{d}) \). Then \( \kappa_{AD} \leq (\kappa_A^*)^3 \).

Remark 7. Note that (2) can be made into an LP by taking the logarithms of all variables.

In the following, let \( \hat{d} \) always denote the optimal value of \( d \) for (2). This optimization problem can be solved in polynomial time using LP techniques. There also exists an equivalent combinatorial description for the rescaling, which will yield additional insights.

Theorem 8 ([9, Theorem 2.5]). Consider a weighted directed graph \( G \) defined by \( V(G) := [n], E(G) := \{(i, j) \in [n]^2 \mid i \neq j, \hat{\kappa}_{ij} > 0\} \), and \( w_G(e) := \hat{\kappa}_{ij} \) for all \( e \in E(G) \). Define \( \hat{t} \) as the maximum geometric mean of a cycle in \( G \), that is

\[
\hat{t} := \max \left\{ \left( \sum_{e \in C} w_G(e) \right)^{\frac{1}{|C|}} \mid C \text{ is a cycle in } G \right\}.
\]

Let \( H \) be a weighted directed graph defined by \( V(H) := V(G) \cup \{r\}, E(H) := E(G) \cup \{(r, i) \mid i \in [n]\} \), and \( w_H(e) := \begin{cases} 
\log \hat{t} - \log \hat{\kappa}_{ij} & \text{for } e \in E(H) \text{ s.t. } r \notin e, \\
0 & \text{for } e \in E(H) \text{ s.t. } r \in e.
\end{cases} \)

Let \( s_i \) denote the length of the shortest path from \( r \) to \( i \) in \( H \), and define \( d_i := \exp(s_i) \)
for all \( i \in [n] \). Then, it holds that \( d = \hat{d} \).

Remark 9. Note that while \( H \) has negative edge weights, it has no negative cycles, because \( \hat{t} \) is defined as the maximum geometric mean of weights over all cycles in \( G \). The graph \( H \) has the same cycles as \( G \), so for every cycle \( C \) of \( H \) it must hold that \( |C| \log \hat{t} \geq \sum_{(i,j) \in C} \log \hat{\kappa}_{ij} \). Therefore, the Bellman-Ford shortest path algorithm can be used to determine the length of the shortest path from \( r \) to \( i \) in \( H \), and, in particular, the values \( s_i \) are well-defined.

### Finding Circuits

What remains is to describe how to find circuits \( C_{ij} \) that include both \( i \) and \( j \). Since we only consider connected matroids, such circuits always exist. We will now describe the algorithm from [9, Theorem 2.14] to find these circuits.

Suppose we have a basis \( B \) of \( M(A) \). We start by computing the set of all fundamental circuits \( \mathcal{F}_B \) with respect to \( B \). If there is a circuit \( C \in \mathcal{F}_B \) with \( i, j \in \mathcal{F}_B \), we set \( C_{ij} := C \) and we are done with this pair \( i, j \).

Otherwise, we construct an undirected graph \( G \) with \( V(G) = B \) and edges between elements that are contained in a common fundamental circuit. Because \( M(A) \) is connected, so is \( G \). First assume \( i, j \notin B \), and recall that, for an element \( e \notin B \), \( C(B, e) \) denotes the fundamental circuit of \( e \) with respect to \( B \). We find a shortest path in \( G \) between any vertex
in the set $C(B, i) \setminus \{i\}$ and any vertex in the set $C(B, j) \setminus \{j\}$. This can be represented by the sequences $V := \{v_1, v_2, \ldots, v_{t+1}\} \subseteq [n] \setminus B$, $v_1 = i$, $v_{t+1} = j$, and $U = \{u_1, u_2, \ldots, u_t\} \subseteq B$. The set $V$ corresponds to vertices of $G$ and the set $U$ to edges of $G$. More precisely, we set $v_i := b$ for some $b \in B$ such that the $i$-th edge on the path connects vertices in $C(B, b)$.

It is shown in [9] that the set $S := (B \setminus U) \cup V$ contains a unique circuit $C$ that contains all $v_i$’s, including $i$ and $j$.

If $\{i, j\} \cap B \neq \emptyset$, assume w.l.o.g. that $i \in B$. Then, for any choice of $V = \{v_1, v_2, \ldots, v_{t+1}\}$ and $U = \{u_1, u_2, \ldots, u_t\}$ as before such that $i \in C(B, v_1)$ and $i \notin C(B, v_t)$ for $t > 1$, the unique circuit in $(B \setminus U) \cup V$ also contains $i$. If both $i, j \in B$, we proceed in a similar manner with $j$. For more details see the original algorithm description [9].

### 2.2.2 Finding a unique circuit in a set $S$

It remains to show how to find the unique circuit in $S$ together with its elementary vector. There are standard tools of linear algebra that can be utilized for this task, but [9] also provides an alternative solution. We provide both of these solutions here because comparing their efficiency and suitability in practical settings will be an important topic of Section 3.2.

The first method is to solve the linear system $A_S \mathbf{x} = \mathbf{0}$ using any generic linear equation solver, and pronounce the result to be the unique circuit contained in $S$. This approach is justified by the next Lemma:

**Lemma 10.** Let $A \in \mathbb{R}^{n \times m}$ and let $S \subseteq [n]$ such that $M(A_S)$ contains a unique circuit $C$. Then, $\text{Ker}(A_S)$ has dimension 1. Let the basic vector of $\text{Ker}(A_S)$ (unique up to scaling) be $v$. Then, $v$ is an elementary vector for $C$ and $C = \text{supp}(v)$.

**Proof.** We will only prove that $\dim(\text{Ker}(A_S)) = 1$, the rest of the lemma statement follows in a straightforward way. Let $C$ be a circuit of $M(A_S)$ and let $v_1$ be its elementary vector. For contradiction, suppose that $\dim(\text{Ker}(A_S)) \geq 2$. That implies that there exists another vector $v_2 \in \text{Ker}(A_S)$ that is linearly independent from $v_1$.

Let $i$ be any element such that $i \in C$. If $(v_2)_i = 0$, define $v_3 := v_2$. Otherwise, let $v_3 := v_2 - \frac{(v_1)_i}{(v_2)_i} v_1$. In both cases, $i \notin \text{supp}(v_3)$. Because $v_3 \in \text{Ker}(A_S)$, $\text{supp}(v_3)$ is a dependent set in $M(A_S)$. Let $C'$ be any inclusion-wise minimal dependent subset of $\text{supp}(v_3)$.

Because $i \in C$ and $i \notin C'$, it must hold that $C \neq C'$, which is a contradiction with the claim that $M(A_S)$ has exactly one circuit. 

The second approach is advantageous if the task is solved repeatedly for the same matrix but different subsets $S$, which is exactly our situation. It requires converting the matrix $A$ into the row-reduced echelon form (RREF) $A' = (I_m | H)$, but this step is independent of the choice of $S$ and only has to be done once in the course of the algorithm:

**Proposition 11** ([9, Inside proof of Theorem 2.14]). Define an elementary vector $g$ for a circuit $C$ using the following procedure: Set $g_{v_1} := 1$. Using Gauss-Jordan elimination, convert $A$ to the form $A' = (I_m | H)$. Now for each $t \in [t]$, the row of $A'_S$ corresponding to $u_t$ contains only two nonzero entries: $A'_{u_t v_1}$ and $A'_{u_{t+1} v_t}$. Propagate the value $g_{v_t}$ by assigning $g_{v_1}, g_{v_2}, \ldots, g_{v_{t+1}}$. Uniquely extend $g$ to the indices in $B \setminus S$ for any basis $B$ so that $g \in \text{Ker}(A)$. Then, $g$ is an elementary vector for $C$ and it holds that $C = \text{supp}(g)$. 


2.2.3 LP Dataset

To evaluate algorithms related to LP, we will use two publicly available collections of instances. The first and older one is Netlib [13]. In order to widen the set of instances considered here, we also consider the larger and more modern repository MIPLIB [14], which contains mixed integer programs (MIP). Because this work only directly applies to LPs and not MIPs, we consider the LP relaxations of MIPLIB instances.

2.2.4 Problem Format

Both Netlib and MIPLIB problems are in MPS format, which is the industry standard for linear programming problems. Without going into unnecessary detail, let us mention that MPS is very similar to a sparsely represented modelling LP form. Reading the problem file into a convenient representation will be discussed later. Most of the considered problems are very sparse and therefore using a sparse representation is crucial.

3 Algorithm Implementation

We use SageMath [21] for the implementation of the algorithm. SageMath uses Python and can be used directly from Python code, which gives the advantage of a variety of available scientific packages. SageMath itself has support for working with both densely and sparsely represented matrices over both exact and inexact fields, working with graphs, and some support for working with matroids. These capabilities make SageMath a good fit for our goal, which is exploratory rather than aiming for practical efficiency.

Despite Python being relatively slow, this is likely not a major bottleneck since a large portion of the run-time of our implementation is spent inside library functions that have heavily optimized low-level implementations. Thus, the potential performance gain of re-implementing the same algorithm in a lower-level language is unlikely to be large.

3.1 Linear Programming Considerations and Challenges

When reading an LP instance represented in the MPS format, we deal with two considerations. First, how to even read MPS and handle it efficiently? Second, the MPS format is akin to the modelling LP form representation, but our algorithm assumes the standard form of LP. A direct solution would be to perform the polynomial conversion from modelling to the standard form; we point out that we can do a bit better by stopping the conversion earlier.

3.1.1 Reading The Problem Matrix from MPS

Several Python libraries allow reading LPs from the MPS format. The problem with most of these libraries is that it is not easy to access the problem matrix itself. In order to avoid writing an MPS parser from scratch, we utilize the open-source scikit-glpk 2 Python package. This package provides Python bindings for GLPK and allows to read the problem into modelling form as sparse SciPy matrices and vectors, which can be easily converted to sparse matrices and vectors native to SageMath.

---

2 Available from https://github.com/mckib2/scikit-glpk
Prior to our work, this package was not able to read all MPS files that are used in Netlib and MIPLIB. To allow efficient reading all of the problems, we needed to make some changes to the library, which we contributed back to the original package\(^3\).

The first modification of the library was fixing a bug caused by earlier modifications of the library that rendered the MPS reading and writing capabilities defunct. The second modification addressed the fact that the MPS reader inside GLPK sometimes internally represents constraints bounded from both sides, i.e. \(b_1 \leq a^\top x \leq b_2\). As this does not directly correspond to a constraint in modelling form, the library did not support reading the problems whose internal GLPK representation contained such a constraint. We have modified the library to split any such constraint into the two constraints \(b_1 \leq a^\top x\) and \(b_2 \geq a^\top x\). Lastly, we have significantly improved the library performance for working with sparse matrices. The original implementation used an intermediate dense representation when combining two matrices into one and only then converted the resulting matrix back into a sparsely represented one. By using a sparse implementation for the combining step, we avoid the need for the costly dense conversion.

### 3.1.2 Solving LP Form Mismatch

The standard procedure to convert an LP in modelling form to an LP in standard form turns inequality constraints into equality constraints and turns generic bounds into non-negativity bounds. Here we show that converting the bounds to the form \(x \geq 0\) is not necessary and it suffices to arrive at the more general form of *box constraints* \(l \leq x \leq u\) for vectors \(l, u \in (\mathbb{R} \cup \{-\infty, +\infty\})^n\) satisfying \(l \leq u\), since this step does not change the value of \(\kappa\) nor the value of \(\kappa^*\). We call an LP \(\{\min c^\top x \mid Ax = b, l \leq x \leq u\}\) to be in semi-standard form.

Our main result here is the following:

\begin{itemize}
  \item \textbf{Theorem 12.} Let \(A\) be a problem matrix of an LP in semi-standard form and \(A'\) a problem matrix for the same LP converted to standard form. Then, \(\kappa_A = \kappa_{A'}\) and \(\kappa^*_A = \kappa^*_{A'}\).
\end{itemize}

We will break down the proof of Theorem 12 into several simpler statements.

The textbook algorithm for converting an LP in semi-standard form to standard form only uses the following operations:

\begin{itemize}
  \item \textbf{(O1) Take a variable} \(x_i\), and express it as \(x_i = x_i^+ - x_i^-\) for new variables \(x_i^+, x_i^- \geq 0\). Let, without loss of generality,
  \[
  A = \begin{pmatrix}
  \bar{A} & & \\
  \vdots & A_{i+} & \\
  \vdots & & \vdots \\
  \end{pmatrix}, \quad
  x = \begin{pmatrix}
  \vdots \\
  \bar{x} \\
  \vdots \\
  x_i \\
  \end{pmatrix}.
  \]
  The problem matrix after this operation is then
  \[
  A' = \begin{pmatrix}
  \bar{A} & & \\
  \vdots & A_{i+} & -A_{i-} \\
  \vdots & & \vdots \\
  \end{pmatrix}.
  \]
\end{itemize}

\(^3\) https://github.com/mckib2/scikit-glpk/pull/24
https://github.com/mckib2/scikit-glpk/pull/25
https://github.com/mckib2/scikit-glpk/pull/23
(O2) Take a bound $\pm x \geq l$ and turn it into a constraint. If we assume the same layout of $x$ as in (O1), the problem matrix after this operation is

$$A' = \begin{pmatrix}
\tilde{A} & A_{x_i} \\
0 & \cdots & 0 & \pm 1 & 1
\end{pmatrix}.$$ 

We will show that performing neither operation changes the values $\kappa, \kappa^*$. By induction it then follows that performing the conversion algorithm also does not change these values.

\textbf{Lemma 13.} $\kappa_A \geq 1$.

\textbf{Proof.} Let us choose any $C \in M(A)$. For all $i,j \in \binom{C}{2}$, both values $\kappa^A(C)$ and $\kappa^A(C)$ are considered when determining the value of $\kappa_A(C)$. Because $\kappa^A = \frac{1}{\kappa^*}$, it must be the case that $\max\{\kappa^A(C), \kappa^A(C)\} \geq 1$. Therefore, the maximum over all pairs of $i,j$ is also greater than 1, meaning that $\kappa_A(C) \geq 1$ for any circuit $C$. By taking maximum over all circuits, we get the lemma statement.

\textbf{Lemma 14.} Performing operation (O1) does not change the value of $\kappa$.

\textbf{Proof.} Let $A, A'$ be problem matrices before and after performing the operation respectively. Firstly we will show the inequality $\kappa_A \leq \kappa_A'$. Let $C \in \mathcal{C}(M(A))$ be the circuit that $\kappa_A = \kappa_A(C)$ and let $g^C = (\overline{g}^C, g_i)^\top$ be its elementary vector. Define the vector $g^{C'} := (\overline{g}^C, g_i, 0)^\top$. For this vector it holds that

$$\begin{pmatrix}
\tilde{A} & A_{x_i} \\
0 & \cdots & 0 & \pm 1 & 1
\end{pmatrix} \begin{pmatrix}
\vdots \\
\vdots \\
\vdots \\
g_i
\end{pmatrix} = Ag^C = 0.$$ 

We will show that $C' = \text{supp}(\overline{g}^{C'})$ is a circuit of $M(A')$. From the above it directly follows that $C'$ is a dependent set. For contradiction assume that there exists a dependent set of $M(A')$ denoted by $C'' \subset C'$. Let $(\overline{g}^{C''}, 0)^\top$ be an elementary vector for this set. Then, $\text{supp}(\overline{g}^{C''}) \subset C'$ is an independent set of $M(A')$, which contradicts that $C'$ is a circuit.

Because $g^{C'}$ has the same non-zeroes as $g^{C}$, we have $\kappa_A(C) = \kappa_A(C')$. Thus,

$$\kappa_A = \kappa_A(C) = \kappa_A(C') \leq \kappa_A'.$$

For the other inequality, consider any $C' \in \mathcal{C}(M(A'))$ with its elementary vector $g^{C'}$. Because the last two columns of $A'$ are linearly dependent, only two scenarios occur for $g^{C'}$:

1. Either $g^{C'} = (\overline{g}^{C'}, g_i, 0)^\top$ or $g^{C'} = (\overline{g}^{C'}, 0, -g_i)^\top$. In that case, $(\overline{g}^{C'}, g_i)^\top$ is an elementary vector for some vector $C \in \mathcal{C}(M(A))$ and it holds that $\kappa_A(C') = \kappa_A(C)$.

2. Otherwise, $g^{C'} = (0, g_i, g_i)^\top$ Then clearly $\kappa_A(C') = 1$.

For any circuit $C'$ it therefore holds that $\kappa_A(C') \leq \max\{\kappa_A, 1\}$. By using the previously stated Lemma 13, we get the inequality $\kappa_A(C') \leq \kappa_A$.

Consider $C''$ such that $\kappa_A = \kappa_A(C'')$. Then it is holds that

$$\kappa_A = \kappa_A(C'') \leq \kappa_A.$$
Lemma 15. For any $D \in \mathbb{D}$, $\mathcal{C}(M(A)) = \mathcal{C}(M(AD))$.

Proof. Multiplying vectors by nonzero constants in a set of vectors does not change the linear dependence of the set. Therefore, rescaling columns with matrix $D$ does not change dependent sets nor inclusion-wise minimal dependent sets.

Lemma 16. Performing operation (O1) does not change the value of $\kappa^*$. 

Proof. Let $A, A'$ be matrices as in the previous proof. We will start by showing that $\kappa^*_{A'} \leq \kappa^*_A$. Consider $D \in \mathbb{D}$ such that $\kappa^*_A = \kappa_{AD}$. Define

$$D' := \begin{pmatrix} \bar{D} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & \cdots & 0 & d_k \\ & & & d_{n+1} \end{pmatrix}.$$ 

Consider the circuit $C' \in \mathcal{C}(M(A'D'))$ such that $\kappa_{A'D'} = \kappa_{A'D'}(C')$ with an elementary vector $\mathbf{g}^{C'}$. Using Lemma 15, we again know that this circuit can be of one of two types:

1. If $\mathbf{g}^{C'} = (\mathbf{g}^{C'}, g_i, 0)^\top$ or $\mathbf{g}^{C'} = (\mathbf{g}^{C'}, 0, -g_i)^\top$, then the vector $\mathbf{g}' := (\bar{g}^{C'}, g_i)^\top$ is an elementary vector for a circuit $C \in \mathcal{C}(M(AD))$. Because this vector shares the same nonzero values with $\mathbf{g}^{C'}$, it holds that $\kappa^*_{A'} \leq \kappa_{A'D'}(C') = \kappa_{AD}(C) \leq \kappa_{AD} = \kappa^*_A$.

2. Otherwise, $\mathbf{g}^{C'} = (0, g_i, g_i)^\top$. In that case, $\kappa_{A'D'}(C') = 1$. By Lemma 13, $\kappa^*_{A'} \leq \kappa_{A'D'}(C') = 1 \leq \kappa_{AD} = \kappa^*_A$.

To prove the other inequality, consider the $(n+1) \times (n+1)$ diagonal matrix with positive entries on the diagonal

$$D' = \begin{pmatrix} \bar{D}' & 0 \\ 0 & \ddots & 0 \\ & & & d_{n+1} \end{pmatrix}$$

such that $\kappa^*_A = \kappa_{A'D'}$.

Define $\bar{D} := \bar{D}'$ and consider $C \in \mathcal{C}(M(AD))$ such that $\kappa_{AD} = \kappa_{AD}(C)$ together with the elementary vector $\mathbf{g}^{C}$. Then, the vector $\mathbf{g}^{C'} := (\mathbf{g}^{C}, 0)^\top$ is an elementary vector of a circuit in $M(A'D')$ with $\kappa_{A'D'}(C') = \kappa_{AD}(C)$. Therefore it must hold that $\kappa^*_A \leq \kappa_{AD} = \kappa_{AD}(C) = \kappa_{A'D'}(C') \leq \kappa_{A'D'} = \kappa^*_{A'}$.

Lemma 17. Performing operation (O2) does not change the value of $\kappa$. 

Proof. Let $A, A'$ be matrices as before, $C' \in \mathcal{C}(M(A'))$ such that $\kappa_{A'} = \kappa_{A}(C')$ and its elementary vector $\mathbf{g}^{C'}$. For the elementary vector it must hold that

$$\begin{pmatrix} A & 0 \\ 0 & \pm 1 \end{pmatrix} \begin{pmatrix} \cdots & \mathbf{g}^{C'} \\ \mathbf{g}^{C'} \\ \cdots \end{pmatrix} = \begin{pmatrix} \cdots \\ \mathbf{0} \\ 0 \end{pmatrix}.$$


This can be split into two conditions
\[ \bar{g} C' \in \ker(A) \]
\[ g_{n+1} = \pm g_n. \]

We will show that \( g := \bar{g} C' \) is an elementary vector of a circuit \( C := \text{supp}(\bar{g} C') \in \mathcal{C}(M(A)) \). We already know that \( C \) is dependent. For contradiction let us now suppose that there exists another dependent set \( \hat{C} \subset C \). This circuit can be extended to a circuit \( \hat{C}' \in \mathcal{C}(A') \) such that \( \hat{C}' \subset C' \), contradicting that \( C' \) is a circuit. The set \( C \) is therefore maximal dependent.

Vectors \( g C' \) and \( g \) have the same nonzero absolute values, which yields \( \kappa_A(C) = \kappa_{A'}(C') \). Consequently,
\[ \kappa_{A'} = \kappa_{A'}(C') = \kappa_A(C) \leq \kappa_A. \]

For showing the opposite inequality, it suffices to observe that every elementary vector \( g \) for a circuit \( C \in \mathcal{C}(A) \) can be uniquely extended to an elementary vector \( g C' \) for a circuit \( C' \in \mathcal{C}(A') \) such that \( \kappa_A(C) = \kappa_{A'}(C') \). Specifically for \( C \) such that \( \kappa_A = \kappa_A(C) \), it then follows that
\[ \kappa_A = \kappa_A(C) = \kappa_{A'}(C') \leq \kappa_{A'}. \]
\[ \blacksquare \]

**Lemma 18.** Performing operation \((O2)\) does not change the value of \( \kappa^* \).

**Proof.** We again denote the matrices \( A \) and \( A' \) before and after performing the operation. First let us show that \( \kappa_{A'}^* \geq \kappa_A^* \). Let \( D' \) be a diagonal matrix with positive numbers on the diagonal such that \( \kappa_A^* = \kappa_{A'}^* \). The scaled matrix can be written as
\[
A'D' = \begin{pmatrix}
A\bar{D}' & 0 \\
0 & \cdots & 0 & \pm d_n & 0 \\
0 & \cdots & \end{pmatrix}
\]

For any elementary vector \( g C' = (\bar{g} C', g_{n+1})^\top \) of a circuit in \( \mathcal{C}(M(A'D')) \) it must hold that
\[ \bar{g} C' \in \ker(A\bar{D}')} \]
\[ g_{n+1} = \pm \frac{d_n}{d_{n+1}} g_n. \]

Let \( C \in \mathcal{C}(A\bar{D}) \) be a circuit such that \( \kappa_{A\bar{D}}(C) = \kappa_{A\bar{D}} \) with an elementary vector \( g \) \( C \). Define 
\[ g C' := (g, \pm \frac{d_n}{d_{n+1}} g_n)^\top \] and observe that \( C' := \text{supp}(g C') \in \mathcal{C}(M(A'D')) \).

Because \( g C' \) contains all the non-zero elements that \( g \) contains, it must be the case that \( \kappa_{A'D'}(C') \geq \kappa_{A\bar{D}}(C) \), leaving us with
\[ \kappa_{A'}^* = \kappa_{A'D'} \geq \kappa_{A'D'}(C') \geq \kappa_{A\bar{D}}(C) = \kappa_{AD} \geq \kappa_A^*. \]

For the other direction, consider \( D \in D \) such \( \kappa_A^* = \kappa_{AD} \). Choose \( D' \) such that
\[
D' := \begin{pmatrix}
\bar{D} & 0 & 0 \\
0 & \cdots & 0 \\
0 & \cdots & 0 \\
\end{pmatrix}
\]
Consider the circuit $C' \in \mathcal{C}(M(A'D'))$ such that $\kappa_{A'D'}(C') = \kappa_{A'D}$ together with its elementary vector $\bar{g}^{C'} = (\bar{g}^{C'}, g_{n+1})^\top$. Plugging into the conditions above, it must hold that

$$\bar{g}^{C'} \in \text{Ker}(AD)$$
$$g_{n+1} = \pm g_n.$$

We can observe that $\bar{g}^{C'}$ is an elementary vector of a circuit $C \in \mathcal{C}(M(AD))$. The vectors $g^{C'}$ and $\bar{g}^{C'}$ have the same nonzero absolute values, therefore $\kappa_{A'D'}(C') = \kappa_{AD}(C)$. By putting everything together, we get

$$\kappa_A' = \kappa_{AD} \geq \kappa_{AD}(C) = \kappa_{A'D'}(C') = \kappa_{A'D'} \geq \kappa_A.'$$  

Combining Lemmas 14, 16, 17 and 18 concludes the proof of Theorem 12.

### 3.2 Practical Aspects of Dealing With the Vector Matroid

To perform computations with the vector matroid and to understand their efficiency, it is important to understand how the relevant operations on vector matroids can be implemented. These details have great implications for the practicality and scalability of the experiments.

#### 3.2.1 Exactness of Computation

When performing computations with real numbers on modern computers, a floating point (FP) number representation is typically used. While this representation is convenient in many cases, practical calculations then suffer from an inherent precision loss. We will show that this causes issues when using FP computations for working with the vector matroid of a matrix. Not only can small numerical differences in values of the matrix completely change the structure of its vector matroid, but they may also arbitrarily change both the values $\kappa$ and $\kappa^*$.

Example 19 shows that inaccuracies introduced by FP computations may lead an algorithm to falsely proceed as if some set $S$ is independent when it in fact contains a circuit.

**Example 19** (Floating-point arithmetics and vector matroid). For $a \in \mathbb{R}$ consider the matrix

$$A := \begin{pmatrix} a & a & 4a \\ 0 & a & a \end{pmatrix}.$$  

Because $3 \cdot A_{+1} + A_{+2} = A_{+3}$, $\{1, 2, 3\}$ is not an independent set of $M(A)$. However, when using floating-point operations $\cdot_F$ and $+_F$, one may get the result

$$3 \cdot_F \begin{pmatrix} a \\ 0 \end{pmatrix} +_F \begin{pmatrix} a \\ a \end{pmatrix} = \begin{pmatrix} 4a + \varepsilon_1 \\ a + \varepsilon_2 \end{pmatrix}$$

for small values of $\varepsilon_1$ and $\varepsilon_2$. This might lead an algorithm relying on floating-point arithmetics to proceed as if all the three columns were linearly independent and to conclude that $\{1, 2, 3\}$ is an independent set of $M(A)$.

The obvious remedy is using exact arithmetic, but this is undesirable for performance reasons. We have tested an alternative approach based on trying to (quickly) find a candidate for a circuit using $p$-bit precision computation for increasing values of $p$, while verifying potential candidate circuits by exact computation. Unfortunately, this approach turned out to be impractical because all attempted choices for the precision parameters led to unreasonably
Experimental Analysis of LP Scaling Methods

high iteration counts. As a matter of fact, the algorithm converged in reasonable time only for very small problem instances. Since we have not found a way to work with the vector matroid using floating-point arithmetics, we use SageMath’s support for exact rational numbers.

Not only is working with the vector matroid without using exact arithmetics tricky to say the least, but we also observe that small changes in $A$ can inflict large changes in both $\kappa$ and $\kappa^*$. This can be dangerous if we obtain a nearly optimal rescaling of a matrix and then try to use it in an inexact context (e.g., run an LP solver that uses FP arithmetics).

Example 20 shows that FP inaccuracies may make an algorithm operate as if $\kappa_A = C$ for an arbitrarily large $C$, despite $\kappa_A = 1$. Moreover, Example 21 shows how not only $\kappa^*$ but also the scaling matrix $D$ is sensitive to inaccuracies.

► Example 20 (Floating-point arithmetic and $\kappa$). For $a, C \in \mathbb{R}^+$ consider the matrix

$$A := \begin{pmatrix} a & a & Ca + \varepsilon \\ a & a & Ca \end{pmatrix}.$$

The only circuit of $M(A)$ is $\{1, 2\}$ and $g = (1, -1, 0)^\top$ is its elementary vector, implying $\kappa_A = 1$. An algorithm based on floating-point numbers could however falsely believe that $\{2, 3\}$ is a circuit with an elementary vector $(0, C, -1)^\top$ and therefore that $\kappa_A = C$.

► Example 21 (Floating-point arithmetics and $\kappa^*$). The vector matroid of matrix

$$A := \begin{pmatrix} \tilde{\kappa} & 1 & \tilde{\kappa} & 0 \\ \tilde{\kappa} & \tilde{\kappa}^2 & 0 & \tilde{\kappa}^2 \\ \varepsilon & 0 & 0 & 0 \end{pmatrix}$$

for $\tilde{\kappa} > 1$ has a single circuit $\{2, 3, 4\}$. The scaling matrix

$$D := \begin{pmatrix} 1 & \tilde{\kappa} \\ \tilde{\kappa} & 1 \\ \varepsilon & 0 \end{pmatrix}$$

gives $AD = \begin{pmatrix} \tilde{\kappa} & \tilde{\kappa} & \tilde{\kappa} & 0 \\ \tilde{\kappa} & \tilde{\kappa}^3 & 0 & \tilde{\kappa}^3 \\ \varepsilon & 0 & 0 & 0 \end{pmatrix}$.

The only circuit then has an elementary vector $g = (0, 1, -1, -1)^\top$, and thus $\kappa_A^* = \kappa_{AD} = 1$.

An algorithm using floating-point numbers may however give results as if $\varepsilon = 0$ due to loss of precision, effectively making $\kappa_A^* = \tilde{\kappa}$.

If we were to use the exactly computed scaling matrix $D$ in an inexact context (assuming $\varepsilon$ is treated as 0), we get $\kappa_{AD} = \tilde{\kappa}^2$. This shows that not only is the value $\kappa^*$ sensitive to inaccuracies, but the same is also true for the scaling matrix $D$.

3.2.2 Relevant Matroid Operations

For the purposes of the algorithm, the following matroid operations are relevant:

- `Components()` splits the vector matroid to matroids for its components,
- `Basis()` finds any basis of the vector matroid,
- `FundamentalCircuits(B)` finds all fundamental circuits with respect to the basis $B$,
- `ElementaryVector(C)` finds an elementary vector for the circuit $C$ and
- `UniqueCircuit(B, U, V)` finds a circuit contained in $(B \setminus U) \cup V$ that is guaranteed to be unique, where $U \subseteq B$, $V \subseteq [n] \setminus B$. 

3.2.3 Utilizing SageMath

SageMath comes with the `LinearMatroid` class which natively provides most of the operations, and allows easy implementation of the remaining ones. If \( M \) is an instance of `LinearMatroid`, then `components()` is implemented by \( M \).\texttt{components}(), `Basis()` is implemented by \( M \).\texttt{basis}() and `FundamentalCircuits(B)` can be implemented by using the method \( M \).\texttt{fundamental_circuit}(B, e) for every non-basic element \( e \).

The operations `ElementaryVector(C)` and `UniqueCircuit(B, U, V)` can be implemented either using the method in Lemma 10 or the one in Proposition 11. In both cases, we will work directly with the matrix \( A \). In the first case this can be done using \( A \).\texttt{matrix_from_columns(C).right_kernel().basis()[0] for `ElementaryVector(C)` and `A.matrix_from_columns(B.difference(U).union(V).right_kernel().basis()[0].support()` for implementing the operation `UniqueCircuit(B, U, V)`.

The latter option is a bit more involved. Firstly, the matrix \( A \) needs to be converted to RREF using Gauss-Jordan elimination. Then, we construct a “vector finding graph” that will allow for efficient implementation of the procedure.

▶ Definition 22 (Vector finding graph). Let \( A \) be a matrix in RREF such that \( M(A) \) is connected, let \( U, V \) and \( B \) be sets as above and let \( S := (B \setminus U) \cup V \). Consider the submatrix \( \bar{A} \) obtained by selecting the columns in \( S \) and the rows where columns in \( U \) have their only nonzero-value (this value is 1). Let \( \bar{A}_{uv} \) denote the indices such that \( \bar{A}_{uv} \) corresponds to \( A_{uv} \).

Then, the vector finding graph of \( A \) is a weighed directed graph \( G \) such that

\[
V(G) := [n],
E(G) := \left\{ (u, v) \in [n]^2 \left| \begin{array}{l}
\text{there is a row } r_{\bar{A}} \text{ in } \bar{A} \text{ whose only nonzero values are in columns } \bar{u} \text{ and } \bar{v} \\
\text{with } w_{\bar{A}}(u, v) := -\frac{\bar{A}_{\bar{r}\bar{u}}}{\bar{A}_{\bar{r}\bar{v}}} \end{array} \right. \right\},
\]

This graph can be constructed in time \( \mathcal{O}(nm) \). Its meaning is that for an elementary vector \( g \) of the unique circuit contained in \( S \) it holds that if \( (u, v) \in E(G) \), then \( \frac{w_u}{g_u} = w_{\bar{g}}((u, v)) \).

The procedure from Proposition 11 can be implemented as follows: start with an empty vector \( w \) and set \( w_v \leftarrow 1 \). Then, run a depth-first search (DFS) from \( v \) in the vector finding graph and for every discovered edge \((i, j)\), set \( w_j \leftarrow w_i \cdot w_{\bar{g}}((i, j)) \). Processing edges in this order guarantees that every edge \((i, j)\) is used only after the value \( w_i \) is known, that everything that can be propagated, and that no value \( g_j \) is computed more that once. Lastly, set values to indices in \( B \setminus S \) such that \( w \in \ker(A) \). This is done by performing the matrix-vector multiplication \( w_{B \setminus S} \leftarrow -(Aw)_{B \setminus S} \), which is a relatively simpler operation than solving a generic system of linear equations as in Lemma 10.

▶ Remark 23. It suffices to convert the matrix \( A \) to RREF only once in the course of the run of the algorithm and then reuse it for all calls of `ElementaryVector` and `UniqueCircuit`.

Upon inspection of the source code of SageMath, we have discovered that its vector matroid implementation cannot leverage matrix sparsity. In particular, the matrix is converted to an internal dense representation when constructing the vector matroid object. Because matrices of problems in Netlib and MIPLIB usually have many zeroes, representing them densely has great impact on both memory and runtime of the operations. It is therefore natural to explore other options that work with the matrix directly in its sparse representation.
3.2.4 Sparsity and RREF

While it might seem sufficient to modify SageMath’s algorithm’s internals to work with the sparse representation of $A$, there are more fundamental problems that make it impossible to fully avoid the memory blow-up caused by converting to a dense representation. The root of the problem lies in needing the RREF of matrix $A$. Even if $A$ is sparse, converting it to RREF it is often not the case and the representation becomes inefficient. This is known as matrix fill-in and has been studied for example by Brayton et al. [6]. Because of this, the memory savings (and consequently time savings) that motivated the use of sparse matrices are not directly possible.

Note that while we can avoid using RREF explicitly by using the alternative method in Lemma 10, the SageMath implementation of vector matroids nevertheless echelonizes the matrix $A$ internally. It might be possible to reorder the columns of $A$ in a way which minimizes the fill-in, however, we are only aware of strategies for minimizing fill-in when computing LU and QR decompositions, but not for computing RREF.

3.2.5 Comparison of the Methods

The choice of a suitable implementation of the UniqueCircuit method depends on the available computational resources. The generic method of Lemma 10 based on sparse operations is a good fit for scenarios where available memory is limited. The specialized method of Proposition 11 internally uses a dense representation (due to SageMath’s implementation of vector matroids) and exhibited higher memory usage during our tests, but had the benefit of considerably shorter run times.

We have opted for the specialized method; we believe that a practical implementation of the scaling algorithm should, in the context of the UniqueVector method, focus either on optimizing sparsity, or avoiding RREF altogether. Nevertheless, the generic and “natively” sparse method can conceivably be more practical for extremely memory-demanding problems.

3.3 Other Implementation Concerns

Besides working with the vector matroid, the algorithm also solves the maximum geometric mean problem for the pairwise imbalance measure digraph, and the single-source shortest path problem for finding the rescaling diagonal values.

3.3.1 Maximum Geometric Mean Problem

The maximum geometric mean problem can be converted to the minimum geometric mean problem in a straightforward way, and then further reduced to the minimum arithmetic mean problem by taking logarithms of all edge weights. This well-known problem can then be solved by Karp’s algorithm [16].

We have not found an implementation of Karp’s algorithm in SageMath, so we opted for our own implementation in plain Python. It may be possible to significantly speed up this step of the algorithm by choosing a lower-level language implementation, however, this procedure only represents a very small portion of the total runtime.

At this point in the algorithm we no longer need to use exact arithmetic; we will formally argue why this is not a problem later (Lemma 24).
3.3.2 Single-source Shortest Path Problem

The graph in which we need to find shortest paths from \( u \) to all other vertices \( v \) contains negative edges, so the natural choice is to use the Bellman-Ford algorithm. This is implemented in SageMath as the method `shortest_paths(u, by_weight=True, algorithm="Bellman-Ford_Boost")` of SageMath graph objects, and it internally calls a Boost implementation of Bellman-Ford which does not support weights that are SageMath-specific infinite-precision rational numbers.

By using inexact arithmetic, the shortest paths can only change by a very small amount. Because the shortest path lengths are exactly the diagonal elements of the obtained rescaling matrix, this causes us to have found a slightly different rescaling. The following lemma shows that this rescaling cannot be much worse than the one found when using exact values.

\[\text{Lemma 24.} \quad \text{Let } \alpha \geq 1, \text{ and let } D \in \mathbb{D} \text{ and } D' \in \mathbb{D} \text{ satisfy } D'_{ii} \in \left[\frac{1}{\alpha} D_{ii}, \alpha D_{ii}\right] \text{ for every } i \in [n]. \text{ Then, } \kappa_{AD'} \leq \alpha^2 \kappa_{AD}.\]

Proof. Consider the circuit \( C \) and indices \( i, j \) such that \( \kappa_{AD'} = \kappa_{ij}^{AD'}(C) \). Let \( g^{AD} \) and \( g^{AD'} \) be the elementary vectors corresponding to \( C \) such that \( g^{AD} \in \text{Ker}(AD) \) and \( g^{AD'} \in \text{Ker}(AD') \). Then,

\[
\kappa_{AD'} = \kappa_{ij}^{AD'}(C) = \left| \frac{g_j^{AD'}}{g_i^{AD'}} \right| \leq \frac{\alpha |g_j^{AD}|}{\alpha^{-1} |g_i^{AD}|} \leq \alpha^2 \kappa_{AD}. \]

Even though the original graph is guaranteed to not contain negative cycles, inexact arithmetics may introduce slightly negative cycles. In that case, the Boost implementation fails. For this reason, we increase all edge weights by a value \( \varepsilon \) to compensate for such inaccuracies, thus making the shortest path up to \( n \cdot \varepsilon \) longer.

4 Performance Comparison

Our main result is the evaluation of the performance of LP solvers both on the original problem and the problem after applying the approximate optimal rescaling. Note that while Dadush et al. [9] give some performance guarantees (indeed, that is their primary motivation), their relevance in our case is somewhat limited. Firstly, all claims in [9] are relevant for the interior point LP algorithm of Vavasis and Ye [23], but none of the tested algorithms directly use this method and, to the best of our knowledge, no theoretical claims have been made about the effects of the rescaling for other algorithms.

The second thing to note is that for problems with large \( \kappa^* \), the rescaling may be very far from an optimal one. Still, the potentially weak guarantee is only a worst-case one, and the algorithm may do well on real-world instances nevertheless.

Lastly, despite the potential pitfalls of working with the circuit imbalance measure using non-exact arithmetic explained earlier, we also evaluate the performance of non-exact (floating-point) LP algorithms. We do so for completeness and because the vast majority of use cases of LP are in FP (not exact) arithmetic.

4.1 Exact Solvers

Some evaluated solvers can be configured to find the solution using exact arithmetic. However, even these solvers read its input from MPS files and at least some of them parse the numeric values in it into floating-point numbers and only then into an exact data type. Because of issues outlined earlier, a naive use of an exact solver will lead to problems with precision loss.
We propose the following workaround. Due to the properties of the IEEE 754 floating-point number arithmetic standard, multiplying an FP number by a (possibly negative) power of two does not cause precision loss if the mathematically correct result can be represented as an IEEE 754 FP number. Therefore, if we restrict the elements of our rescaling matrix to be such powers of two, the rescaled problem as loaded by the exact solver will be mathematically precisely the original problem with scaled columns of the problem matrix.

The next lemma shows that rounding down the rescaling matrix obtained by our algorithm to integral powers of two gives a rescaling not much worse than the original one:

**Lemma 25.** Let \( \hat{D} = \text{Diag}(\hat{d}) \in \mathbb{D} \). Define \( \hat{d}_i \) to be the nearest smaller integral power of two than \( d_i \) for every \( i \in [n] \) and let \( \hat{D} := \text{Diag}(\hat{d}) \). Then, \( \kappa_{\hat{A}\hat{D}} \leq 2 \cdot \kappa_{AD} \).

The proof of this lemma is almost identical to the proof of Lemma 24.

### 4.2 Experimental Methodology

We have measured solving times for both original and rescaled instances using the open-source solvers GLPK and SCIP [1] and using the proprietary solver Gurobi [15]. In all cases, garbage collection was disabled during the measuring process to keep it from interfering with the measurement. We exclude parsing time from the measurement. We did our best to disable any built-in scaling so as to not interfere with our scaling. We repeat every measurement 10 times and then consider the arithmetic mean to mitigate faulty measurements.

We include problem instances that were solved in a very short time (under 0.0003 seconds). As was pointed out by the reviewers, the time ratio for these instances might not be very meaningful. We are thankful for this remark and we will make a more careful selection of the measured results in a future journal version of this paper.

Our main interest is the relative effect the rescaling has on the solver run time. Let \( \tau_o \) be the time that a solver needs to solve the original instance, and \( \tau_r \) the time the solver needs to solve the rescaled instance. Define the time ratio of the rescaling as \( \rho := \left( \frac{\tau_r}{\tau_o} \cdot 100 \right) \% \). For floating-point solvers, we measure \( \tau_r \) using the precise rescaling, while exact solvers we use the rescaling rounded to powers of two as described in Subsection 4.1.

To measure \( \tau_o \), an MPS file created from the problem converted to semi-standard form is used in contrast to using the original MPS files. This is done so that the difference between the two tested problems is solely in column scaling and structural differences of the matrices can not affect performance.

The software versions used to perform the measurements were GLPK v4.65 (floating point simplex and interior point methods, and an exact simplex method), Gurobi v10.0.2 (floating-point default method), SCIP v8.0.3 (floating-point simplex) and an exact version of SCIP v8.0.0.14 (exact simplex). All solver settings unrelated to scaling or exact arithmetic are set to their default values. We thank the reviewers for noting that this makes it more difficult to draw conclusions about the results of Gurobi and SCIP, as they may use both simplex and interior point methods as appropriate for every instance. Because changing this setting would require running new experiments with different solver settings, we will consider this in a future journal version of this paper.

We used a computational cluster where each machine is equipped with AMD EPYC 7532 CPUs which have SPECfp2017 score of 6.9 per core. The solvers were limited to use a single CPU core.

---

4 Available from https://github.com/scipopt/scip/tree/exact-rational
4.3 Measurement Results

We managed to compute the approximate rescaling for a total of 142 problem instances of MIPLIB and Netlib. We ran the measurements for all solution methods and all problem instances. We exclude measurements that have not finished in the allotted time.

The best average time ratio was attained for GLPK using an interior-point method with a value of 132%. The highest number of problem instances with time ratio under 100% was attained for Gurobi with 51 (36%) said problem instances.

The lowest (best) achieved time ratio is 7.6%, meaning that applying the rescaling led to a 13× speed-up of the solver. On a similar note, for 112 of the problem instances the rescaling had a time ratio smaller than 100% for at least one of the solution methods, providing a speed-up in the runtime.

We have not observed strong correlations between speed-ups for the various solvers.

Detailed results of the experiment can be found in Figure 1. Columns of the scatter plot correspond to individual instances. In addition to time ratios for all solving methods, we have included the best and the mean time ratio for every problem instance.

4.3.1 Improvement of \( \kappa \)

Besides possible performance benefits, it is also natural to ask about the improvement of the circuit imbalance measure. Ideally, we would report statistics about the ratio \( \kappa_A / \kappa_{AD} \), however, due to the intractability of computing these quantities, we resort to comparing the improvement with respect to the set of circuits \( C' \) computed by the algorithm. Specifically, recall that \( \hat{\kappa}_A \) is a lower bound on \( \kappa_A \) based on \( C' \), and let \( \hat{\kappa}_{AD} \) be the largest imbalance of any circuit in \( C' \) with respect to \( AD \); thus, \( \hat{\kappa}_{AD} \) gives a lower bound on \( \kappa_{AD} \). Figure 2 shows the relative improvement (wrt. \( C' \)) \( \hat{\kappa}_A / \hat{\kappa}_{AD} \), and the (absolute) new estimate \( \hat{\kappa}_{AD} \).

Again, note that this is a lower bound.

Admittedly, comparing two lower bounds gives a fairly weak claim. It is however unclear whether there is much of an alternative: one could try generating (or sampling) many circuits to enlarge \( C' \) and get better bounds, but short of computing the \( \kappa \)'s exactly, it is impossible to show (useful) definite bounds on the quality of the rescaling.
References


Algorithms for Gradual Polyline Simplification

Nick Krumbholz
University of Konstanz, Germany

Stefan Funke
University of Stuttgart, Germany

Peter Schäfer
University of Konstanz, Germany

Sabine Storandt
University of Konstanz, Germany

Abstract

Displaying line data is important in many visualization applications, and especially in the context of interactive geographical and cartographic visualization. When rendering linear features as roads, rivers or movement data on zoomable maps, the challenge is to display the data in an appropriate level of detail. A too detailed representation results in slow rendering and cluttered maps, while a too coarse representation might miss important data aspects. In this paper, we propose the gradual line simplification (GLS) problem, which aims to compute a fine-grained succession of consistent simplifications of a given input polyline with certain quality guarantees. The core concept of gradual simplification is to iteratively remove points from the polyline to obtain increasingly coarser representations. We devise two objective functions to guide this simplification process and present dynamic programs that compute the optimal solutions in \(O(n^3)\) for an input line with \(n\) points. For practical application to large inputs, we also devise significantly faster greedy algorithms that provide constant factor guarantees for both problem variants at once. In an extensive experimental study on real-world data, we demonstrate that our algorithms are capable of producing simplification sequences of high quality within milliseconds on polylines consisting of over half a million points.

2012 ACM Subject Classification Theory of computation → Computational geometry; Theory of computation → Design and analysis of algorithms

Keywords and phrases Polyline simplification, Progressive simplification, Fréchet distance

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.19

Funding This work was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project ID 251654672 – TRR 161.

1 Introduction

Polyline simplification is the process of reducing the complexity of linear structures while ensuring that the output still resembles the input. There is a variety of applications for polyline simplification, including data compression [17, 14], noise reduction in movement trajectories [15, 16], and visualization of linear data on maps [19]. In particular in interactive cartographic visualizations, there are often large data volumes that need to be rendered efficiently. For example, the front-end of a trajectory search engine must be able to display huge amounts sets of trajectory data over a map layer, and map rendering tools that allow customization (e.g. rayshader [18]) must accommodate selected features quickly, see Figure 1.

Simplifying the data to the desired level of detail prior to rendering is a suitable step to reduce the data complexity, thereby reducing the rendering time and avoiding visual clutter.

Formally, a polyline \(P = p_1, p_2, \ldots, p_n\) is defined as a sequence of points and the induced straight line segments between consecutive points. In the classical polyline simplification problem, the input is a polyline \(P\), a distance measure \(d_X\), and a threshold \(\varepsilon > 0\).
Figure 1 Left: 3D map (created with rayshader [18]), overlaid with linear features extracted from OSM, namely waterways (blue), primary and secondary roads (yellow and white), and a GPS track (red). The waterways and roads consist of roughly 240,000 points in total, the GPS track of 1481 points. Right: Simplifications of the GPS track with 10%, 5% and 1% of the original points in red (original track for visual comparison in gray).

A line segment $S_{ij}$ — also called shortcut — between points $p_i$ and $p_j$ with $j > i$ is called valid if $d_X(P[i,j], S_{ij}) \leq \varepsilon$, where $P[i,j]$ refers to the subpolyline of $P$ from $p_i$ to $p_j$ and $d_X(P[i,j], S_{ij})$ denotes the shortcut error. The goal of polyline simplification is to compute a minimum-sized path from $p_1$ to $p_n$ that only uses valid shortcuts. The endpoints of these shortcuts define the simplified polyline $P' \subseteq P$. Typical similarity measures for polylines are the Hausdorff distance $d_H$ and the Fréchet distance $d_F$. The polyline simplification problem can be solved optimally in $O(n^2)$ for $d_H$ [8] and in $O(n^2 \log n)$ for $d_F$ [22].

However, a single simplification of the given input is often not sufficient for the application scenario. If the linear features are to be displayed on a zoomable digital map, a simplified representation is needed for each zoom level. There are three main guidelines the resulting sequence of simplifications should adhere to:

- **Complexity.** The level of detail should decrease monotonically when zooming out.
- **Conformity.** The shape should be sufficiently preserved across all zoom levels.
- **Consistency.** Once simplified away, polyline points should not reappear later.

To generate such simplification sequences, the concept of progressive line simplification problem is used in [5]. Here the input is a polyline $P$ and a sequence of distance thresholds $\varepsilon_1 < \varepsilon_2 < \cdots < \varepsilon_k$. The task is to compute a valid polyline simplification for each $\varepsilon_i$, where the simplification for $\varepsilon_i$ has to be a subset of the points chosen for the simplification for $\varepsilon_{i-1}$ for all $i > 1$. This subset constraint ensures visual consistency between the simplifications. The optimization goal is to find a sequence of simplifications with the smallest number of shortcuts accumulated over all simplification levels. It was shown in [5] that the problem can be solved to optimality in $O(n^3k)$ for both $d_H$ and $d_F$. However, determining an appropriate sequence of distance thresholds for progressive line simplification is a non-trivial task that demands the user to have knowledge about the polyline structure. For example, it can happen that several consecutive $\varepsilon$ values in the sequence induce the same or a very similar simplification, which only adds computational complexity but no further visual benefits. Thus, a continuous version was discussed in [5], where the distance threshold sequence does
not need to be specified, since all possible \( \varepsilon \) values at which the simplification might change are taken into account. However, the algorithm for computing the respective simplification sequence has a rather impractical running time of \( O(n^{5}) \).

In this paper, we introduce the _gradual line simplification (GLS) optimization problem_. GLS is based on the concept of iterative point removal to generate a fine-grained and consistent simplification sequence. The objective function ensures that the shape of polyline is well preserved throughout the simplification process. We will discuss two problem variants and show that both can be solved very efficiently in theory and practice. In contrast to progressive line simplification, GLS does not require the user to specify distance thresholds, but simplification sequences with any desired level of detail and with any number of zoom levels can be extracted based on a single GLS solution.

### 1.1 Related Work

The concept of generalization of geometrical objects for visualization on zoomable maps is well-established in cartography [23, 21, 7, 4]. There are many different incarnations as, e.g., object shrinking or fading, feature elimination, smoothing, or merging of different objects. For linear structures displayed on maps such as country borders, rivers, roads, or GPS-based movement trajectories, polyline simplification is typically applied to obtain coarser representations [25, 11].

Optimal algorithms for polyline simplification rely on first constructing a shortcut graph and computing a shortest link path therein [8, 22]. However, heuristics typically avoid the expensive shortcut graph construction phase and instead select points to be preserved in the simplification in a greedy fashion. Iteratively discarding points as long as the shortcut distance threshold is not violated is a wide-spread algorithm design paradigm here. For example, the heuristic by Visvalingam and Whyatt [24] works by always removing the point from the polyline which together with its neighbors forms a triangle of smallest area. The algorithm runs in \( O(n \log n) \) time. The famous Douglas-Peucker algorithm [9] proceeds the other way around by selecting promising points one by one until the shortcut distance threshold is obeyed. The Douglas-Peucker algorithm can be implemented to run in \( O(n \log n) \) under \( d_H \) and in \( O(n^2) \) under \( d_F \) [13]. Further examples of iterative simplifications are the algorithms by Wang-Müller [26] or Zou-Jones [10]. All algorithms following the design paradigm of iterative point removal generate a consistent simplification sequence as a by-product. But the existing methods do not provide quality guarantees for the resulting sequence.

To cater for multiple zoom levels, hierarchical simplification was studied e.g. in [19]. In their setup, they have multiple non-intersecting polylines and a sequence of distance thresholds \( \varepsilon_1 < \cdots < \varepsilon_k \) and they seek to find for each \( \varepsilon_i \) simplifications for all polylines that obey the respective distance threshold and additionally remain intersection-free. However, they do not take consistency into account.

Polyline simplification algorithms can also be useful for simplifying polygons or more complex structures as polygonal subdivisions and graphs. For polygons, a single point is chosen as start and end point of the polyline (one could also iterate over all points in that role and record the best result). For subdivisions and graphs, the simplification problem can be reduced to a set of independent polyline simplifications by retaining all nodes with a degree of three or more [11]. The process of iterative point removal and shortcut insertion can however also be applied to graphs with arbitrary node degrees, as explored in [3] for consistent and continuous road network simplification. But the focus there is also on fast heuristics to deal with large inputs.
1.2 Contribution

We introduce gradual line simplification (GLS) as a formal optimization problem and provide several theoretical and practical results. We consider two problem variants, one in which we aim to minimize the maximum error of any introduced shortcut (max-error) and one in which we minimize the sum over all shortcut errors (sum-error). We show that optimal max-error or sum-error solutions can be computed in time \( O(n^3) \) and space \( O(n^2) \). This result applies for shortcut errors being measured by Fréchet or Hausdorff distance. Under the Fréchet distance, we prove that a simple greedy algorithm simultaneously approximates the max-error by a factor of 2 and the sum-error by a factor of 4. This greedy algorithm can be implemented to run in \( O(n^2) \) time. But to achieve this running time heavy machinery is used. Therefore, we also present a more practical greedy variant that only relies on simple methods and allows to trade running time against quality. Furthermore, we propose output-sensitive algorithms to extract any simplification of desired size from the GLS sequence. The underlying data structure has linear size and can be computed in linear time. This allows a user to easily select simplification subsequences of desired complexity. Finally, we provide an extensive experimental study in which we evaluate our methods on real GIS data. As the main outcome, we observe that the proposed approximation algorithms are orders of magnitude faster than the exact algorithms while producing close-to-optimal results.

2 Preliminaries and Definitions

In this section, we provide the formal GLS problem definition, and describe notation and basic algorithmic building blocks that will be used throughout the paper.

2.1 Shortcutting and Gradual Simplification

The core concept of GLS is to obtain a sequence of \( n-1 \) consistent simplifications of size \( n, n-1, \ldots, 2 \) by iterative point removal from the input polyline. Formally, this removal is referred to as point shortcutting as defined below.

\textbf{Definition 1 (Point Shortcutting).} Given a polyline \( P = p_1 \ldots p_n \) and a point \( p_i \) with \( i \in \{2, \ldots, n-1\} \), the shortcutting operation removes \( p_i \) from \( P \) and creates \( S_{i-1,i+1} \).

Based on this notion, GLS is defined as follows.

\textbf{Definition 2 (Gradual Line Simplification (GLS)).} Given a polyline \( P = p_1 \ldots p_n \) and a point shortcutting order \( \pi : \{p_2, \ldots, p_{n-1}\} \to \{1, \ldots, n-2\} \), we denote the sequence of polyline simplifications \( P_0, \ldots, P_{n-2} \) as gradual simplification of \( P \) where \( P_0 = P \) and \( P_i \) is derived from \( P_{i-1} \) by shortcutting the point \( p_j \) with \( \pi(p_j) = i \). The total set of shortcuts that results from the respective point shortcutting operations is denoted by \( S_\pi(P) \).

Figure 2 illustrates GLS on a small example instance.

According to the definition, the final simplification always consists solely of the shortcut from the start to the end point of the input polyline, that is \( P_{n-2} = p_1, p_n = S_{1n} \). We observe that any shortcutting order results in a consistent polyline simplification sequence as we clearly ensure \( P_i \subset P_j \) for any \( i > j \). Furthermore, the simplification sequence is the most fine-grained one possible among all consistent simplification sequences, i.e., it contains the maximum number of distinct simplifications.

To guide the gradual simplification process, we devise two different optimization problems. Both aim at simplification sequences that preserve the shape of the input polyline.
Figure 2 Example polyline (black) with \( n = 29 \) nodes. A shortcutting order of the inner points is indicated by the green labels. Additionally, the resulting simplifications \( P_{10} \) (solid blue, containing only points with a label higher than 10) and \( P_{20} \) (dashed purple, containing only points with a label higher than 20) are displayed.

Definition 3 (Max-error GLS). Given a polyline \( P = P_1, \ldots, p_n \) and a distance measure \( d_X \), find a shortcutting order \( \pi \) that minimizes the maximum shortcut error under \( d_X \), that is, \( \min \max_{S \in S_\pi(P)} \epsilon(S) \).

Definition 4 (Sum-error GLS). Given a polyline \( P = P_1, \ldots, p_n \) and a distance measure \( d_X \), find a shortcutting order \( \pi \) that minimizes the sum of shortcut errors under \( d_X \), that is, \( \min \sum_{S \in S_\pi(P)} \epsilon(S) \).

2.2 Shortcut Error Computation

The GLS problem and the algorithms we will design crucially rely on access to shortcut errors \( \epsilon(S_{ij}) := d_X(P[i,j], S_{ij}) \) under a given distance measure. Throughout this paper, we focus on the Hausdorff distance \( d_H \) and the Fréchet distance \( d_F \). Classical polyline simplification algorithms usually only require an oracle that decides whether a shortcut \( S_{ij} \) for \( 1 \leq i < j \leq n \) has distance at most \( \varepsilon \) to its respective subpolyline \( P[i,j] \). Such an oracle can be implemented for both \( d_H \) and \( d_F \) in time \( O(n) \). However, for gradual line simplification, the actual shortcut errors are relevant. Under \( d_H \), such errors can be computed in \( O(n) \). Under \( d_F \), using the algorithms by Alt and Godau [2], the error can be computed in \( O(n^2) \) in a quite simple fashion, or in \( O(n \log n) \) if parametric search is applied. We will refer to the more expensive variant also as vanilla error computation method. Recently, Buchin et al. [6] introduced a Fréchet distance data structure (FDS) that allows to preprocess a given polyline in time and space \( O(n \cdot k^{4+\delta} + n^2) \) for \( \delta > 0 \) and then answers distance queries from any line segment to a selected subpolyline in time \( O(\frac{n^2}{k} \log^2 n + \log^4 n) \) for some of \( k \in [n] \). We will exploit this data structure with a careful choice of \( k \) to compute relevant shortcut errors faster in our algorithms.
3 Exact Algorithms

We first show that the max-error GLS and the sum-error GLS problem can be solved to optimality in polytime under both $d_H$ and $d_F$. In particular, we design efficient dynamic programs (DP) that require $O(n^3)$ time and $O(n^2)$ space.

The main observation is that we can decompose our problem into subproblems which encode whether or not a certain shortcut is contained in the solution. Let $\varepsilon(S_{ij})$ denote the error of shortcut $S_{ij}$. Further, let $\varepsilon_{\text{max}}(S_{ij})$ denote the max-error of an optimal solution of the gradual simplification problem restricted to $P[i,j]$, which implies that $S_{ij}$ is part of the solution. Similarly, $\varepsilon_{\text{sum}}(S_{ij})$ denotes the optimal sum-error of gradual polyline simplification for $P[i,j]$. Let $p_k$ be the last point shortcut before $p_i$ and $p_j$, i.e. the point whose shortcutting resulted in the insertion of $S_{ij}$. Then the shortcuts (or original segments) $S_{ik}$ and $S_{kj}$ are part of the solution as well, and we have $\varepsilon_{\text{max}}(S_{ij}) = \max\{\varepsilon_{\text{max}}(S_{ik}), \varepsilon_{\text{max}}(S_{kj}), \varepsilon(S_{ij})\}$ and $\varepsilon_{\text{sum}}(S_{ij}) = \varepsilon_{\text{sum}}(S_{ik}) + \varepsilon_{\text{sum}}(S_{kj}) + \varepsilon(S_{ij})$. Based on these formulas, we can construct the solution set recursively starting with $S_{1n}$, which always has to be contained in any solution. However, we don’t know the value of $k$ if we go top-down. But this can easily be overcome by iterating over all possible $k$ with $i < k < j$ and picking the minimum resulting max- or sum-error. We can store already computed solutions for $S_{ij}$ in a look-up table to avoid redundant computations. This results in the following dynamic program: We allocate an $n \times n$ table and initially set all entries to 0. In cell $c_{ij}$ with $i < j$, we store $\varepsilon_{\text{max}}(S_{ij})$ or $\varepsilon_{\text{sum}}(S_{ij})$, respectively. The table cells are filled by using the above formulas. As we always need access to all shortcuts of smaller hop length to get the correct results, we consider the cells sorted increasingly by $j - i$.

**Theorem 5.** The DP approach solves max-error and sum-error GLS in time $O(n^3)$ under $d_H$ and $d_F$, respectively, using quadratic space.

**Proof.** The created table has a space consumption of $\Theta(n^2)$. The running time is determined by the time needed to fill the $O(n^2)$ cells. To get the correct cell value $c_{ij}$, we first need to compute $\varepsilon(S_{ij})$ and then iterate over all values $k$ between $i$ and $j$ and check cells $c_{ik}$ and $c_{kj}$. The latter part can be accomplished in constant time per considered cell and hence takes $O(n)$ in total. The computation of $\varepsilon(S_{ij})$ depends on the distance measure. For $d_H$, it takes time $O(n)$. For the Fréchet distance, as discussed above, this would take $O(n \log n)$ when using the parametric search technique. However, based on the FDS by Buchin et al. with a choice of $k \in \Theta(n^{1/3})$, preprocessing the polyline and determining all potential shortcut errors can be accomplished in $o(n^3)$ using $O(n^2)$ space. Thus, we spend on average $O(n)$ time per cell for both $d_H$ and $d_F$, which amounts to an overall running time of $O(n^3)$. \hfill \triangleright

Figure 3 shows an example where the optimal outcomes for sum-error and max-error coincide. If in the optimal sum-error solution the max-error is assumed by the final shortcut $S_{1n}$, the solution is automatically also optimal for max-error. However, as visible in the example, the max-error is not necessarily assumed by $S_{1n}$ and thus the two problem variants have different solutions in general.

4 Approximation Algorithms under the Fréchet Distance

The cubic running time and the quadratic space consumption of the exact algorithm might be prohibitive in practice, especially when dealing with large inputs. Therefore, we next investigate approximation algorithms with better performance. We will present constant-factor approximations for both max-error and sum-error GLS under $d_F$. Our algorithms rely on a well-known lemma by Agarwal et al. [1], rephrased below in our terminology and illustrated in Figure 4.
Figure 3 Example polyline (grey) and optimal gradual simplification (blue and red shortcuts) for sum-error and max-error. The red polyline corresponds to the respective simplification of the input polyline after half the points have been shortcut.

Lemma 6. Given a polyline $P$, consider a shortcut $S_{ij}$ with error $\varepsilon$. Then for any shortcut $S_{ab}$ with $i \leq a < b \leq j$, its error under $d_F$ is bounded by $d_F(P[a,b], S_{ab}) \leq 2\varepsilon$.

We remark that for any distance measure where the factor between the error of $S_{ij}$ and $S_{ab}$ is bounded by a constant, the algorithms we describe below will yield constant-factor approximations for max-error and sum-error GLS. As this does not apply to $d_H$, though, our approximation results do not transfer to the Hausdorff distance. However, the Fréchet distance is usually deemed the more appropriate polyline distance measure anyhow, as it takes the course of the polyline explicitly into account while the Hausdorff distance treats the polyline as unordered set of contained points.

4.1 2-Approximation for Max-Error GLS

We first consider the max-error GLS problem. In the following lemma, we show the somewhat surprising result that any shortcutting order produces a maximum shortcut error within a factor of 2 of the optimal one under the Fréchet distance.

Lemma 7. Any order $\pi$ is a 2-approximation for max-error GLS under $d_F$.

Proof. According to the GLS definition, we know that for any $\pi$ we have $S_{1n} \in S_\pi(P)$. Thus, $\varepsilon(S_{1n})$ is a lower bound for the optimum value $OPT$. Lemma 6 implies that for any possible $S_{ab} \in S_\pi(P)$ it holds $\varepsilon(S_{ab}) \leq 2\varepsilon(S_{1n})$ and hence $\varepsilon(S_{ab}) < 2OPT$. As this applies for any shortcut error it clearly also applies to the maximum shortcut error in any set $S_\pi(P)$.

The lemma illustrates that solely focusing on the max-error does not provide enough guidance to obtain practically useful simplification sequences. Thus, we will from now on focus on the sum-error variant.

Figure 4 Illustration of Lemma 6.
4.2 4-Approximation for Sum-Error GLS

In the following we present and analyze a simple greedy algorithm for sum-error GLS: Given the current simplification \( P_t \), we always choose the next point \( p \in P_t \) to remove by selecting the one whose shortcutting results in the currently smallest shortcut error among all candidates. We will now prove that this strategy yields a constant-factor approximation algorithm and subsequently investigate its running time and space consumption.

\[ \textbf{Theorem 8.} \ \textsc{Greedy} \text{ is a 4-approximation algorithm for sum-error GLS under } d_F. \]

\textbf{Proof.} Let \( P \) be a polyline of size \( n \). Let \( S_1, \ldots, S_{n-2} \) be the shortcuts created by the greedy algorithm in their insertion order and let \( \pi_1, \ldots, \pi_{n-2} \) denote the points in \( P \) according to their contraction order. Further, let \( S_1^*, \ldots, S_{n-2}^* \) be the shortcuts inserted based on an optimal point ordering. We use \( P_t \) or \( P_t^* \) to refer to the subpolyline by \( S_t \) or \( S_t^* \), respectively. As before, we use \( \varepsilon(S) \) to denote the shortcut error of a shortcut \( S \).

We construct an assignment of shortcuts inserted by the greedy algorithm to optimal shortcuts. In particular, we assign shortcut \( S_t \) to \( S_t^* \) if the following two conditions are met:

- At the moment before \( \pi_i \) is shortcut by the greedy algorithm, there are at least three points in \( P_t^* \) that are not yet shortcut, including \( \pi_i \).
- Index \( j \) is the smallest index in the optimal ordering for which the above property holds.

Note that the assignment is well-defined, as we have \( S_{n-2}^* = s_{1n} \) and the first condition is always true for \( S_{n-2}^* \) as we never shortcut its endpoints.

We now show that if \( S_t \) is assigned to \( S_t^* \) it holds \( \varepsilon(S_t) \leq 2\varepsilon(S_t^*) \). This applies because if at least three points of \( P_t^* \) are not shortcut before the shortcutting of point \( \pi_i \), we know that shortcutting the middle of these three points would result in a shortcut for a subpolyline of \( P_t^* \). According to Lemma 6, the error of any such shortcut is upper bounded by \( 2\varepsilon(S_t^*) \).

As the greedy algorithm selects the next point to shortcut based on the minimum possible induced error at the current stage, we thus conclude that \( \varepsilon(S_t) \leq 2\varepsilon(S_t^*) \) has to hold as well.

Let now \( c_j \) be the number of shortcuts \( S_t \) assigned to a particular shortcut \( S_t^* \) in the optimal solution. Then the greedy sum-error can be upper bound by \( \sum_{i=1}^{n-2} \varepsilon(S_t) \leq \sum_{j=1}^{n-2} c_j \cdot 2\varepsilon(S_t^*) \).

To complete the proof, we will argue that \( c_j \leq 2 \) for all \( j = 1, \ldots, n-2 \). Assume now for contradiction that there exists a shortcut \( S_t^* \) to which we assigned at least three greedy shortcuts. Let the respective points that resulted in these shortcut insertions in that order be \( q_1, q_2, q_3 \). Based on the assignment criterion, we have \( q_1, q_2, q_3 \in P_t^* \). Furthermore, there need to be three points that are not shortcut yet at the point greedy considers \( q_3 \) in order for the respective shortcut to get assigned to \( S_t^* \). Hence we have at least also \( q_4, q_5 \in P_t^* \) that are shortcut after \( q_3 \). Now we consider the point \( p^* \) that was shortcut by \( S_t^* \) in the optimal solution. At that moment, there were only the points \( p_l, p^*, p_r \) left in \( L_j^* \), where \( p_l \) is the left endpoint of \( S_t^* \) and \( p_r \) the right one. Hence the shortcuts (or original line segments) \( S_t = (p_l, p^*) \) and \( S_r = (p^*, p_r) \) exist in the optimal solution. If those are shortcuts, their indexneeds to be smaller than \( j \), as they were constructed before \( S_t^* \). Now if we have \( q_1, q_2, q_3, q_4, q_5 \in P_t^* \), at least three of them have to be contained in either the subpolyline belonging to \( S_t \) or \( S_r \). W.l.o.g. assume it is \( S_t \). That automatically implies that \( S_t \) is indeed a shortcut and not an original segment. Now if we shortcut the \( q_i \) with smallest index \( i \) in \( S_t \) (which implies \( i \leq 3 \)), we assign \( S_t \) to \( S_t^* \), as \( S_t^* \) has a smaller index than \( S_t^* \), and there are at least three not yet shortcut points in the respective subpolyline including \( q_4 \). This contradicts our claim that \( S_t \) is assigned to \( S_t^* \). We therefore conclude that \( c_j \leq 2 \) holds.

Accordingly, we get \( \sum_{j=1}^{n-2} c_j \cdot 2\varepsilon(S_t^*) \leq \sum_{j=1}^{n-2} 2 \cdot 2\varepsilon(S_t^*) = 4 \sum_{j=1}^{n-2} \varepsilon(S_t^*) = 4 \cdot OPT \).
According to Lemma 7 and Theorem 8, a greedy ordering provides simultaneously a 2-approximation for max-error and a 4-approximation for sum-error.

Lemma 9. \textsc{Greedy} runs in $O(n^2 \log n)$ using linear space.

Proof. The greedy algorithm needs to compute linearly many shortcut errors. The initial errors of shortcuts $S_{ii+2}$ for $i = 1, \ldots, n - 2$ can be computed in $O(1)$. Then, after selecting the next point to shortcut, only the values for its two former neighbors need to be updated which amounts to $O(n)$ non-trivial error computations. If we use parameteric search for shortcut error computation, the overall running time is $O(n^2 \log n)$. As only linearly many shortcut errors need to be stored at any time, the space consumption is linear.

For a faster implementation of the greedy approach, we can again exploit the FDS by Buchin et al., however, at the cost of an increased space consumption.

Lemma 10. \textsc{Greedy} can be implemented to run in $O(n^2)$ time using quadratic space.

Proof. Recall that FDS requires a preprocessing time of $O(n \cdot k^{3+\delta} + n^2)$ and has a query time of $O\left(\frac{n}{k^3} \log^2 n + \log^4 n\right)$ for some $k \in [1, n]$. As we need to issue $O(n)$ shortcut error queries, the total time of \textsc{Greedy} based on FDS can be expressed as $O(n \cdot k^{3+\delta} + n^2 + \frac{n^2}{k} \log^2 n + n \log^4 n)$. If we choose $k$ to be slightly smaller than $n^{1/3}$, e.g. $k = n^{(3-\delta)/3}$, then all summands are in $O(n^2)$. Selecting $n - 2$ times the one with minimum error among the current $O(n)$ shortcut candidates can clearly also be accomplished in $O(n^2)$.

This implementation of the \textsc{Greedy} algorithm thus matches the space consumption of the DP for exact sum-error GLS computation but reduces the running time by a factor of $n$.

4.3 A More Practical Greedy Approach

Both greedy variants described in Lemma 9 and Lemma 10 rely on sophisticated search techniques or data structures and are cumbersome to implement. If we use the vanilla approach by Alt and Godau [2] with a running time of $O(n^3)$, the implementation is much easier. However, the resulting running time is in $O(n^3)$ and thus too slow for large input polylines. Therefore, we now discuss another variant of the greedy algorithm tailored to practical implementation. That is, we only want to use easy-to-implement subroutines but get competitive running times nevertheless.

To achieve this goal, we first observe that the greedy algorithm does not necessarily require the knowledge of the precise shortcut errors of all candidates, but only needs to identify the currently smallest one. Thus, it would suffice to identify a threshold $\varepsilon$ such that one candidate has a smaller error and the others a larger one. This concept would allow us to only use the decision oracle whether a shortcut exhibits an error of at most $\varepsilon$ as subroutine (just like in classical polyline simplification). Binary search over the range of possible $\varepsilon$ values until we have the desired division of candidates sounds appealing. But this approach suffers from the issue that if two shortcuts exhibit very similar errors, then the number of search steps might be huge; potentially even incurring a larger running time than the naive implementation using vanilla shortcut error computation. But if we abort the binary search after a fixed number of search steps, then there might be still several candidates in the remaining interval and the ratio between their errors could be unbounded (in particular if the lower bound of the current search interval is still zero). If that error ratio is unbounded, the greedy algorithm loses its approximation guarantee. Thus, we need a sensible stopping criterion for the search that allows us to bound the number of steps as well as the approximation factor. The next theorem shows that this is indeed possible.
\textbf{Theorem 11.} \textit{GREEDY} can be implemented to run in $O(c \cdot n^2 \log_b n)$ time using linear space, with an approximation factor of $(4b + \frac{1}{n})$ for any $b, c > 1$.

\textbf{Proof.} Let $\varepsilon = \varepsilon(S_{1n})$ be the shortcut error of the final shortcut, which is necessarily a part of any GLS. Accordingly, we have $OPT \geq \varepsilon$, and we know by Lemma 6 that all shortcut errors are bounded by $2\varepsilon$.

In every round, we then proceed as follows with the two newly created shortcut candidates: We first invoke the decision oracle for $2\varepsilon$. As long as the oracle returns true, i.e., that the shortcut errors is below or equal to the threshold, we divide the current bound by $b$ and repeat. We abort the process if the oracle returns false or if the bound drops below $\frac{\varepsilon}{n}$ for some $c \geq 1$. This happens after at most $c \cdot \log_b(n) + \log_b(2) \in O(c \log_b n)$ rounds. Then, we choose one of the shortcut candidates with smallest known error upper bound for the next insertion. The overall running time is in $O(c \cdot n^2 \log_b n)$ as there are less than $n$ rounds, and in each round we conduct at most two searches with $O(c \log_b n)$ steps each, where a single search step consists of invoking the decision oracle which takes linear time. The value of $\varepsilon$ to initialize the searches can be computed using the vanilla approach in $O(n^2)$.

To prove the approximation guarantee, we first consider the shortcuts that were inserted because their error is at most $\frac{\varepsilon}{n}$. As there are less than $n$ shortcuts in total, the summed error is upper bounded by $\frac{n \varepsilon}{n} \leq OPT \cdot n^{1-c}$. For the shortcuts that came with a larger error, we identified an interval with its lower bound being within a factor of $b$ of the upper bound. Now if we perform the same assignment of shortcuts in the greedy solution to shortcuts in the optimal solution as described in Theorem 8, we still assign at most two greedy shortcuts to each optimal shortcut $S_j^*$. For the greedy shortcut $S_i$ this implies that at the time of its insertion there was also a candidate shortcut with error at most $2\varepsilon(S_j^*)$. As the binary search approximates the error within a factor of $b$, we thus can guarantee that $\varepsilon(S_i) \leq 2b \cdot \varepsilon(S_j^*)$. Plugging this into the sum formula from the proof of Theorem 8, we end up with an accumulated error of at most $4b \cdot OPT$. Combined with the $OPT \cdot n^{1-c}$ error bound from the first part of the analysis, we hence get an overall approximation guarantee of $(4b + \varepsilon) \cdot OPT$ where $\varepsilon = n^{1-c}$ can be made as small as desired.

The theorem allows a simple implementation of the greedy algorithm that trades running time against approximation quality. As $b$ is the crucial parameter for this trade-off, we will from now on refer to this greedy variant as \textit{b-GREEDY}. For practical application, it is also worthwhile to use a priority queue (PQ) data structure (e.g. a min-heap) to take care of the selection of the next best shortcut to insert, reducing the running time with the naive sweep from $O(n^2)$ to $O(n \log n)$. Here, we use the upper bounds of the error intervals as key. While the overall running time is still dominated by the upper bound searches in the worst case, the reduced selection time still has an impact in practice.

5 \hspace{1em} \textbf{Indexing and Output-Sensitive Extraction}

The result of GLS of an input polyline $P$ is a sequence consisting of $n - 1$ simplifications. For some applications the sequence will be used as a whole (e.g. morphing between representations [20], or continuous zooming [23]), for other applications we might only want to use a certain subsequence at a time (e.g. when dealing with a fixed number of zoom levels) or even only select a single simplification of desired size.

Thus, we would like to have fast access to individual simplifications. Given a polyline $P$ and a shortening order $\pi$, we want to index this data such that for given $k$, the simplification consisting of $k$ line segments (that is the one after contracting $n - k - 1$ points) can be
The authors themselves recommend to rather use the vanilla variant with a running time \( O(k) \). A simple way to achieve this is to store all \( n - 1 \) simplifications explicitly. But this requires quadratic space. We note that it suffices for given \( k \) to retrieve the set of points \( p \) with \( \pi(p) \geq n - k \) in their order of occurrence in \( P \). The corresponding shortcuts are simply the line segments between consecutive points in the ordered set. This gives rise to the following simple data structure: We store the points of \( P \) in the given order in an array and augment them with their \( \pi \)-values. We also store the position of the point contracted last, that is, the point with \( \pi(p) = n - 2 \). Furthermore, for each point \( p_i \), we store the index of the point \( p_l \) in \( P[i, i - 1] \) with largest \( \pi \)-value smaller than that of \( p_i \) such that for all points \( p' \in [p_l + 1, p_l - 1] \), we have \( \pi(p') < \pi(p_l) \). Similarly, we store the point \( p_r \) in \( P[i + 1, n] \) with largest \( \pi \)-value smaller than that of \( p_i \) such that for all points \( p' \in [p_i + 1, p_r - 1] \), we have \( \pi(p') < \pi(p_r) \). If \( p_l \) or \( p_r \) do not exist, we store a default index instead. Clearly, the data structure has linear space consumption. It can also be computed in linear time based on an algorithm for 1D range maxima queries [12].

In a query, where \( k \in [n] \) is provided by the user, we use the constructed data structure as follows. If \( k = 1 \), we simply return the first and the last point in the array. For \( k > 1 \), we initialize a double linked list \( P' \) with \( p_1, p_n \) and then insert the point \( p \) with \( \pi(p) = n - 2 \) into the middle of this list. As its position information is precomputed, it only takes constant time to access the point and to look up its stored values for \( p_l \) and \( p_r \). If \( \pi(p) \leq n - k \), we insert \( p_i \) as left neighbor of \( p \) in \( P' \) and proceed recursively. Similarly, if \( \pi(p_r) \leq n - k \), we insert it as right neighbor of \( p \) in \( P' \) and then proceed recursively as well. Clearly, inserting a new element in \( P' \) as neighbor of a currently considered element takes constant time. In total, we insert \( k - 1 \) additional elements after initialization. For each element or point \( p \), we spend constant time on checking the \( \pi \)-values of the point itself as well as \( p_l \) and \( p_r \), respectively. Thus, overall the extraction time is \( O(k) \), making the described algorithm output-sensitive.

We further remark that if the user wants to extract a simplification subsequence from the data structure with sizes \( k_1 < k_2 < \cdots < k_s \), a total running time of \( O(k_s \log k_s) \) can be achieved no matter the value of \( s \). We simply need to ensure in the query for the simplification of size \( k_s \) that whenever we consider multiple candidate points with \( \pi(p) \leq n - k_s \), we add the one with highest \( \pi \) value first. In that way, we faithfully reconstruct the whole suffix of the simplification sequence down to the one of size \( k_s \) and thus can easily return any requested subsequence. Keeping track of the maximum takes \( O(k_s \log k_s) \) in total when using a max-heap data structure. Of course, if only a subsequence is needed, one could attempt to directly solve the optimization problem where given a size sequence \( k_1 < k_2 < \cdots < k_s \), the goal is to find a consistent simplification sequence adhering to the given sizes with smallest maximum or summed shortcut error. The advantage of instrumenting GLS is that it serves as a lightweight data structure from which any desired sequence can be extracted quickly, e.g. to cater for different devices with varying rendering capabilities or to dynamically adapt the zoom level. Consistency is always guaranteed by construction.

### 6 Experimental Study

For experimental evaluation, we implemented the proposed algorithms for GLS in C++. Experiments were conducted on a single core of an AMD Ryzen processor at 4.2 GHz.

All algorithms were implemented using the Fréchet distance to measure shortcut errors. The best theoretical running time bounds for \( \text{DP} \) and \( \text{GREEDY} \) rely on FDS. However, this data structure is very sophisticated and was not implemented so far. Also, the parametric search variant by Alt and Godau to compute shortcut errors in \( O(n \log n) \) is quite involved. The authors themselves recommend to rather use the vanilla variant with a running time
Table 1 Running times of the implemented algorithms for GLS when shortcut errors under \( d_F \) are computed in \( \mathcal{O}(n^2) \). Approximation factors for max- and sum-error GLS are provided below.

<table>
<thead>
<tr>
<th></th>
<th>max-DP</th>
<th>sum-DP</th>
<th>GREEDY</th>
<th>( b )-GREEDY</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>( \mathcal{O}(n^4) )</td>
<td>( \mathcal{O}(n^4) )</td>
<td>( \mathcal{O}(n^3) )</td>
<td>( \mathcal{O}(n^2 \log b n) )</td>
</tr>
<tr>
<td>apx max</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>apx sum</td>
<td>-</td>
<td>1</td>
<td>4</td>
<td>( 4b + \varepsilon )</td>
</tr>
</tbody>
</table>

of \( \mathcal{O}(n^2) \) for practical purposes. The vanilla variant is easy to implement and numerically robust. We hence only use this method for shortcut error computation in our experiments. Table 1 provides an overview of the resulting running times of our proposed algorithms. Furthermore, we implemented four simple baselines:

- **ORDER**: Use the given input point order as order for gradual simplification.
- **RAND**: Use a random permutation of the points.
- **HOPS**: Choose the next shortcut to insert to be the one where the corresponding subpolyline of \( P \) contains the smallest number of points, i.e. has smallest hop-distance. The idea is that shortcuts that span fewer points are more likely to induce a small Fréchet error than shortcuts that span long parts of the polyline.
- **AREA**: Choose the next shortcut to insert to be the one with the smallest triangle area, i.e. the triangle formed by the new shortcut and the two shortcuts it is replacing [24].

The first three baseline algorithms can all be implemented to run in linear time. **AREA** takes \( \mathcal{O}(n \log n) \) time. By virtue of Lemma 7, all of them provide a 2-approximation for max-error GLS. But none of them comes with an approximation guarantee for sum-error GLS.

### 6.1 Comparative Performance of GLS Algorithms

To test our algorithms on a diverse set of polylines, we used openly available trajectories from OSM. The database holds almost a million trajectories of varying size. For the experiments, we picked 100 random trajectories of each size (if available) and used different upper bounds on the trajectory size. We always specify along with each experiment which sets were used.

We first conducted comparative experiments on the DP, the GREEDY algorithm, 2-GREEDY, and the four baselines with respect to running time and solution quality. Table 2 provides an overview of the results on instances with up to 1000 points (containing roughly 100000 trajectories). As expected, the DP approach is very slow, taking several minutes already

<table>
<thead>
<tr>
<th></th>
<th>running time</th>
<th>approximation ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(( \mu s ))</td>
<td>sum error</td>
</tr>
<tr>
<td></td>
<td>avg. max.</td>
<td>min. avg. max.</td>
</tr>
<tr>
<td>DP</td>
<td>584 s 2,423 s</td>
<td>1.0 1.3 1.4</td>
</tr>
<tr>
<td>GREEDY</td>
<td>3 s 58 s</td>
<td>1.0 1.5 3.4</td>
</tr>
<tr>
<td>2-GREEDY</td>
<td>897 1,934</td>
<td>1.0 1.7 7.1</td>
</tr>
<tr>
<td>AREA</td>
<td>290 464</td>
<td>1.0 1.5 2.9</td>
</tr>
<tr>
<td>HOPS</td>
<td>272 434</td>
<td>1.5 2.2 4.5</td>
</tr>
<tr>
<td>RAND</td>
<td>233 382</td>
<td>2.5 69.2 197.6</td>
</tr>
<tr>
<td>ORDER</td>
<td>109 227</td>
<td>2.5 69.2 197.6</td>
</tr>
</tbody>
</table>
on small input polylines. \textsc{Greedy} is two orders of magnitude faster than \textsc{DP}, but the exact computation of the Fréchet distance values for the candidates is still quite expensive. 2-\textsc{Greedy} is vastly faster. On the sample instances, all results were computed in less than 2 milliseconds. This yields a speed-up of more than a factor of 1000 over the \textsc{Greedy} algorithm with naive Fréchet distance computation. For max-error GLS our experiments show that the theoretical approximation ratio of 2 is tight and that there is little difference in performance among the algorithms. For sum-error GLS, we observe that all approaches except \textsc{Order} produce reasonable results. \textsc{Greedy} performs best and stays way beyond the theoretical approximation factor of 4. 2-\textsc{Greedy}, with an approximation guarantee of \( \approx 8 \), also performs much better in practice but slightly worse than \textsc{Greedy}. It thus offers a good trade-off between running time and quality.

In Figure 5, we further compare the running time and the solution quality of 2-\textsc{Greedy} and the baselines on larger instances. As the \textsc{DP} was too slow to compute exact solutions, we provide a relative quality comparison to the output of 2-\textsc{Greedy}. As expected, the baseline methods are considerably faster as they do not rely on shortcut error computation. But they also produce results of worse quality. While \textsc{Area} often comes close to the quality of 2-\textsc{Greedy}, there are some notable outliers in \textit{sum error}. \textsc{Hops} is more consistent but also worse than 2-\textsc{Greedy} on almost all instances. We conclude that 2-\textsc{Greedy} is the more reliable approach with reasonable running time costs.

### 6.2 Sensitivity Analysis for \( b \)-\textsc{Greedy}

Next, we strive to investigate the performance of our proposed \( b \)-\textsc{Greedy} algorithm more closely. Above, we used \( b = 2 \) in the comparative evaluation. Now, we study the influence of the parameter \( b \).

Figure 6 depicts the running times and the solution quality for various values of \( b \) ranging from 1.1 to 4. As expected from the theoretical analysis, the running time decreases with growing \( b \) but the solution quality deteriorates slightly. Thus, if in a specific application one aspect is more important than the other, \( b \) can be chosen appropriately.

Setting \( b = 3 \), we obtain running times of around 200 milliseconds on average and a maximum of around 6 seconds for trajectories with over half a million points.
Figure 6 b-GREEDY performance for various values of b. Input sizes ≤ 1000 vertices. Quality relative to optimal = 1.

6.3 Further Engineering of b-GREEDY

The practical running time of the b-GREEDY approach can be further reduced in practice by decreasing the number of calls to the shortcut error oracle. We observe that in particular for shortcut candidates with small shortcut errors, our proposed procedure of starting with an oracle call with error $2\varepsilon(S_{1:n})$ and proceeding to divide the current error by $b$ until the oracle returns false for the first time is quite time-consuming. If we start the procedure with a tighter upper bound on the shortcut error, fewer oracle calls are induced. To get such an upper bound for shortcut $S_{ij}$, we use $d_F(S_{ij}) \leq \max\{d(p_i, p_k), d(p_k, p_j) \mid i < k \leq j\}$. So we simply compute the maximum Euclidean distance of any point in $P[i,j]$ to the shortcut end points $p_i$ and $p_j$ in time $O(j - i)$. As the time to compute this upper bound is equivalent to that of an oracle call, the overall running time decreases even if only a few oracle calls are saved. A second observation that helps to avoid unnecessary oracle calls is that error intervals do not need to be refined all the way down to the backstop $\varepsilon/n$ immediately. Instead, if we consider the set of current shortcut candidates, refinements are only necessary until the best candidate is separated from the others. Thus, upon insertion of a new element in the PQ we issue oracle calls with decreasing error bounds until false is returned, or until we reach the same upper bound as we have for the current top element in the PQ. In the latter case, we refine both of them further until the oracle returns false for one (or both) of them. Thus, especially when the best candidate has a significantly smaller error than the others, oracle calls are saved. We investigate the impact of backstop value and of the proposed

<table>
<thead>
<tr>
<th>$\varepsilon_0$ (m)</th>
<th>2-GREEDY time (ms)</th>
<th>'oracle' calls</th>
<th>% worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>43.5</td>
<td>1961</td>
<td>1.6</td>
</tr>
<tr>
<td>10</td>
<td>53.1</td>
<td>2261</td>
<td>5.1</td>
</tr>
<tr>
<td>1</td>
<td>65.1</td>
<td>2973</td>
<td>10.4</td>
</tr>
<tr>
<td>0.1</td>
<td>80.8</td>
<td>3601</td>
<td>17.8</td>
</tr>
<tr>
<td>1e-3</td>
<td>96.7</td>
<td>4171</td>
<td>27.0</td>
</tr>
<tr>
<td>1e-5</td>
<td>102.0</td>
<td>4447</td>
<td>30.0</td>
</tr>
</tbody>
</table>
engineering on the number of oracle calls for the 2-GREEDY algorithm. Table 3 summarizes our findings. The backstop value $\varepsilon_0$ provides a lower bound for the shortcut error interval refinement process and can be modulated via the choice of $c$ in Theorem 11. As we deal in our experiments with real-world trajectories, we provide $\varepsilon_0$ in meters. As to be expected, the smaller the backstop the larger the running times as more oracle calls need to be issued to differentiate between shortcuts with small errors. As shown in the last column of the table, using improved upper bounds on the shortcut errors and only refining intervals on demand reduces the number of oracle calls by 25-52%. This helps to achieve fast processing times even for huge trajectories with over half a million points.

### 6.4 Showcase Application

As discussed in Section 5, one strength of GLS is that after the solution sequence is computed, an simplification consisting of $k$ segments can be extracted in time $O(k)$ for any given $k$. This is an important feature for dynamic map rendering, where polylines or polygonal shapes are displayed on demand based on user interaction and selection.

In Figure 7, top, the island of Mallorca is shown as represented in OpenStreetMap with roughly 35000 points. When rendering Mallorca as part of a larger map, it is clearly unnecessary to draw it with all these details. So a rendering engine might ask for a simplification using only few hundred points. This can be easily realized by drawing only the last $k$ points in the GLS ordering where $k$ can be chosen completely freely (e.g. depending on the rendering capability of the device). Figure 7 also shows simplifications of Mallorca using different numbers of points. In a zoomed out view, there is hardly any difference between the representation using 35000 points versus the representation using 137 points, which are less than 0.5% of the input. Again, note that such a representation can be obtained using only 137 steps based on our index structure after GLS computation.

---

**Figure 7** Results of gradual simplification for Mallorca with (from top to bottom) 35000, 548, 137, and 11 points.
Figure 8 Comparison of GLS based simplification (top) with naive subsampling (bottom) on the port of Mallorca instance using 4382 points each. In the bottom image, shape distortions and visual artifacts are clearly visible.

A naive method that allows for such fast simplification would be to perform simple subsampling, where only every $n/k$-th point is displayed. While not requiring any preprocessing, the results are considerably worse, as can be seen in a closeup of the port of Mallorca in Figure 8. Here we show a zoomed in view of the port of Mallorca in a simplification consisting of 4,382 points. On top, using GLS, we see that the piers are well preserved, whereas using subsampling as depicted in the bottom picture leads to a heavily distorted shape. Furthermore, using subsampling on different zoom levels violates our requirement of producing consistent simplification sequences. GLS, on the other hand, offers full flexibility in the selection of desired local simplifications and simplification sequences, which are guaranteed to be consistent with each other.

7 Conclusions and Future Work

We proposed the gradual line simplification problem and provided practically useful algorithmic solutions. In particular the $b$-GREEDY algorithm turns out to be an easy to implement and fast approximation algorithm for GLS, which allows to trade running time against quality. Choosing suitable parameters, even huge input lines and line sets can be processed in real time. By providing $n-1$ consistent simplifications of a given input polyline with a single computation at once, GLS caters to applications as continuous zooming or customized level of detail selection.

In future work different objective functions for GLS could be investigated, as e.g. the sum of squared shortcut errors, or the sum over the shortcut errors of each individual simplification. The latter takes the order in which shortcuts are inserted into account, yielding a more intricate optimization problem.
References


8. Wing Shiu Chan and Francis Chin. Approximation of polygonal curves with minimum number of line segments or minimum error. *International Journal of Computational Geometry & Applications*, 1996.


Algorithms for Gradual Polyline Simplification


Targeted Branching for the Maximum Independent Set Problem Using Graph Neural Networks

Kenneth Langedal
University of Bergen, Department of Informatics, Norway

Demian Hespe
Karlsruhe Institute of Technology, Institute for Theoretical Informatics, Germany

Peter Sanders
Karlsruhe Institute of Technology, Institute for Theoretical Informatics, Germany

Abstract

Identifying a maximum independent set is a fundamental NP-hard problem. This problem has several real-world applications and requires finding the largest possible set of vertices not adjacent to each other in an undirected graph. Over the past few years, branch-and-bound and branch-and-reduce algorithms have emerged as some of the most effective methods for solving the problem exactly. Specifically, the branch-and-reduce approach, which combines branch-and-bound principles with reduction rules, has proven particularly successful in tackling previously unmanageable real-world instances. This progress was largely made possible by the development of more effective reduction rules. Nevertheless, other key components that can impact the efficiency of these algorithms have not received the same level of interest. Among these is the branching strategy, which determines which vertex to branch on next. Until recently, the most widely used strategy was to choose the vertex of the highest degree. In this work, we present a graph neural network approach for selecting the next branching vertex. The intricate nature of current branch-and-bound solvers makes supervised and reinforcement learning difficult. Therefore, we use a population-based genetic algorithm to evolve the model’s parameters instead. Our proposed approach results in a speedup on 73% of the benchmark instances with a median speedup of 24%.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms; Computing methodologies → Neural networks

Keywords and phrases Graphs, Independent Set, Vertex Cover, Graph Neural Networks, Branch-and-Reduce

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.20

Supplementary Material
Software (Source Code): https://github.com/KennethLangedal/CutBranching-GNN archived at svh:1:dir:833015a33a533b40c350f3f9849ba2d0e4b99f59
Software (Source Code): https://github.com/KennethLangedal/vc-satreduce-gnn archived at svh:1:dir:f694b7e0ba0bb7ae9bdc88424d6f3a6ff8744047

Funding Kenneth Langedal: Supported by the Research Council of Norway under contract 303404 and NORA National Research School for AI, project number 331723
Peter Sanders: European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement No. 882500)

Acknowledgements We thank Fredrik Manne for all the help and feedback he provided throughout the writing process.

1 Corresponding author

© Kenneth Langedal, Demian Hespe, and Peter Sanders; licensed under Creative Commons License CC-BY 4.0
22nd International Symposium on Experimental Algorithms (SEA 2024).
Editor: Leo Liberti; Article No. 20; pp. 20:1–20:21
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
An independent set of an undirected graph $G = (V, E)$ with vertex set $V$ and edge set $E$, is a subset $I \subset V$ such that the vertices in $I$ are pairwise non-adjacent, i.e. $\forall u, v \in I : \{u, v\} \notin E$. The problem of finding an independent set of maximum cardinality is a fundamental NP-hard problem [15]. Its applications span computer graphics [44], network analysis [41], route planning [24], and computational biology [3, 7]. More applications use its complementary problems minimum vertex cover and maximum clique.

Branch-and-reduce is one of the most successful techniques in both theory [6, 54] and practice [1, 18, 19, 40] to solve the maximum independent set problem (MIS). Branch-and-reduce algorithms combine classical branch-and-bound algorithms with repeated application of so-called reduction rules after every branching step. Reduction rules can remove known parts of the graph that are provably in or out of some maximum independent set or such that a solution for the reduced graph can be lifted to a solution for the original graph. These reduction rules can often drastically reduce the size of the graphs, leading to faster processing times.

In the past, research has mainly been focused on developing reduction rules [2, 5, 8, 20, 49]. Recently, other aspects like search space pruning [40] and branching vertex selection [18] have been shown to have considerable impact on the performance of branch-and-reduce solvers. Some first results using machine learning have also shown to be effective for heuristic (inexact) solvers [26, 32], where vertices that are likely to be part of a solution are predicted.

2 Related Work

In this section, we cover the existing work on branching techniques and introduce graph neural networks. We also give a brief overview of the different learning paradigms used in this area.

2.1 Branching Vertex Selection

The most commonly used strategy for selection a branching vertex for maximum independent set and minimum vertex cover is to select a vertex of maximum degree [1, 14, 19, 40]. Akiba and Iwata [1] compare this strategy to branching on a vertex of minimum degree or a random vertex and show that both of these perform substantially worse. Wang et al. [53] also compare maximum degree branching to other simple heuristics for the weighted version of minimum vertex cover and find maximum degree branching to perform best. Hespe et al. [18] introduce several new branching vertex selection strategies and show that targeting vertices whose removal enables new reductions consistently leads to improved or equal performance compared to branching on the highest degree vertex. They show this result on benchmark instances from the 2019 PACE challenge on vertex cover [12] and the 1993 DIMACS challenge on maximum clique [21].

Most algorithms for maximum clique use a greedy algorithm to compute a vertex coloring and branch on vertices with a high coloring number [47, 51]. Other algorithms use the so-called degeneracy order of the graph [4] or iterative maximum independent set computations [29] to determine the branching order. Some algorithms use MaxSAT encodings to prune the set of branching vertices [29, 30, 31].
2.2 Graph Neural Networks

Graph Neural Networks (GNNs) are machine learning models that process and analyze data structured as graphs [45]. Traditional neural networks are not well-suited for processing graph-structured data since they operate on fixed-sized input vectors. GNNs address these limitations by introducing specialized architectures that handle graph data effectively. One popular GNN architecture is the Graph Convolutional Network (GCN) introduced by Kipf and Welling [25]. At each layer in a GCN, each node aggregates information from its immediate neighbors and combines it with its own data. After this, the information stored in each node is passed through a layer-specific neural network to create the node information for the next layer. This architecture is the same one we use to build our GCN models, and it is defined as follows: Let $A$ be the adjacency matrix for an undirected graph with added self-edges and $D$ be the diagonal degree matrix, where $D_{ii} = \sum_j A_{ij}$. Each vertex has a feature representation that changes at each layer in the model. The length $d$ of the feature representation at layer $l$ is denoted by $d^{(l)}$. Stacking all the feature vectors at the $l$'th layer gives the matrix $H^{(l)} \in \mathbb{R}^{\mid V \mid \times d^{(l)}}$. Finally, as in traditional neural networks, we have the trainable parameters $W^{(l)} \in \mathbb{R}^{d^{(l)} \times d^{(l+1)}}$ and bias $b^{(l)} \in \mathbb{R}^{1 \times d^{(l+1)}}$, in addition to a non-linear activation function $\sigma$, such as ReLU$(x) := \max(0, x)$. With this, we use the following layer-wise propagation rule.

$$H^{(l+1)} = \sigma(D^{-\frac{1}{2}}AD^{-\frac{1}{2}}H^{(l)}W^{(l)} + b^{(l)})$$

In our case, there are no features associated with the vertices. The only input to the problem is the graph itself. Therefore, we set the input dimension $d^{(0)} = 1$ and initialize $H^{(0)}$ with all ones.

To clarify the propagation rule further, the first part of the equation $D^{-\frac{1}{2}}AD^{-\frac{1}{2}}H^{(l)}$ is a message-passing step where each vertex aggregates information from its neighbors. The inclusion of the two $D^{-\frac{1}{2}}$ matrices scales each message with both the sender and receiver’s degree. The rest, $\sigma(H^{(l)}W^{(l)} + b^{(l)})$, is like any traditional neural network. Both $H$ and $W$ are dense matrices where each row in $H$ corresponds to the feature vector of one vertex. The bias term $b$ is a vector added to each row of the output at this layer. See Figure 1 for an illustration of one GCN layer.

Supervised Learning

A widely used technique to train GNNs for combinatorial optimization tasks is supervised learning. Examples of this include the traveling salesperson problem [22], coloring [27], maximum cut [46], and maximum independent set [26, 32]. Given a set of training examples with associated labels or targets, supervised learning aims to adapt the model’s parameters based on these examples and targets. More specifically, using a loss function that measures how well the current parameters align with the target outputs, the problem is to minimize this loss value. Assuming the loss function is continuous and differentiable, we can compute gradients for each parameter and perform gradient descent. For a more in-depth introduction to supervised learning, see, for instance, Goodfellow et al. [16].

Reinforcement Learning

Unlike supervised learning, reinforcement learning does not rely on training examples. Instead, an agent—in this case, a GNN model—explores an environment by taking actions. After choosing an action, the new state is evaluated into a reward and passed back along with the
Figure 1 Illustration showing the layer-wise propagation rule introduced by Kipf and Welling [25]. The first column shows the input graph and input matrix \( H^{(0)} \). Since the input is only the graph itself, the feature matrix \( H^{(0)} \) is initialized with a single 1 for each vertex. The second column shows the message-passing step. The matrix \( T \) is only used in this figure as the temporary result between the message-passing and the dense neural network part. Since the message-passing step uses \( D^{-\frac{1}{2}} AD^{-\frac{1}{2}} \), the output for each vertex depends on both its own degree and the degree of each vertex in its neighborhood. The last column shows the dense neural network part. Here we do a dense matrix multiplication between \( T \) and the trainable parameters at this level \( W^{(0)} \) to get \( H^{(1)} \). Finally, the bias vector \( b^{(0)} \) is also added to each row of \( H^{(1)} \). The activation function is not shown in the illustration but is applied to each element of \( H^{(1)} \) at the very end.

Genetic Algorithms

A far less common technique is to use genetic algorithms to train the model’s parameters. Genetic algorithms make use of biology-inspired mechanisms to evolve a solution over time. These mechanisms include crossover, mutation, and selection [33]. Historically, some contention existed between supervised learning using backpropagation and genetic algorithms [36]. However, the former turned out to be far more successful, especially in light of computer vision. Still, recent results have shown that genetic algorithms can be a competitive alternative to reinforcement learning [13, 43, 50]. In these comparisons, the problem was Atari games or similar benchmarks. To our knowledge, training GNNs using genetic algorithms has yet to be applied to combinatorial optimization tasks.

3 GCN Guided Branching

We now briefly motivate our GCN-based branching strategy before describing how we train and use the models. For motivation, several good arguments exist for using machine learning for branch selection. First, the default strategy used in most solvers – branching on the state to the agent. The goal is to maximize this cumulative reward. This learning paradigm saw a surge in popularity after the success of AlphaGo for the game of Go [48]. It has since also been successfully applied to combinatorial optimization problems with GNNs. Examples of this include the traveling salesperson problem [10, 23], vehicle routing [39], subgraph matching [52], and minimum vertex cover [23].
highest degree vertex – is already a heuristic. Regardless of the strategy used, the correctness of the algorithm is not changed, and the asymptotic running time remains the same. However, we know from previous studies that the choice of branching strategy can have a large impact on overall performance. The benchmark instances typically used are not arbitrary graphs either. For example, the instances from the 2019 PACE challenge include transit graphs, road networks, and social network graphs [12]. It is reasonable to think that there are patterns in this data that a machine-learning model can pick up and learn new branching strategies from.

At a high level, our strategy is to let a GCN model decide the next vertex to branch on. Before each branch, we pass the remaining graph as input to a pre-trained GCN model. The output from the model is a single value for each vertex \((d=1)\) at the last layer. The solver then uses the vertex with the highest value as the next branching vertex.

Training a GCN model to pick the next vertex to branch on is problematic from a machine-learning perspective. Ideally, we would like to do supervised learning with labeled data. However, it is not obvious how to generate such data. Several factors could make one branch better than another. For instance, after branching, the resulting graph could:

- Break apart into multiple connected components
- Yield better lower/upper bounds
- Shrink further due to reduction rules

Furthermore, these benefits may not manifest immediately. Instead, the best strategy may be to perform several seemingly bad branches that eventually lead to early termination. This makes it difficult to find optimal branching sequences, even for small instances. Previous results show that it can be done [38], but we argue that two main issues still remain.

The first is non-unique solutions. As with the independent set problem itself, an optimal solution is not necessarily unique. Also, the intersection between optimal solutions could be empty, meaning a GCN could output an optimal solution while being completely wrong measured against the known target. In their supervised learning approach to the MIS problem, Li et al. [32] addressed this issue by having the GCN model output multiple solutions and using the best one to compute gradients, using hindsight loss. While this does address the problem, it’s not a solution.

The second problem is that following a branching sequence close to the optimal could result in a vastly different execution time compared to the optimal. For example, consider the case where the optimal sequence splits the graph into two connected components. If only one vertex appeared later in the sequence, the graph could remain connected, and the execution time could be worse than the maximum degree branching. The point is that we would be training on a proxy problem, and a model that achieves low loss on the training data would not necessarily perform well in practice. This also distinguishes this problem from the broader branch of prediction algorithms [34], which should be learnable with few examples. For these reasons, we do not attempt supervised learning.

The next logical step would be to use some form of reinforcement learning. The problem of picking the next branch fits nicely into the agent-action-reward cycle. However, the main problem with this approach is the intricate implementations of the existing solvers. Moving between states has the added cost of checking reduction rules, testing connectivity, and searching for lower bounds. It would not make sense to turn off these parts, as it’s precisely this interaction between the branching and other elements of the solver we wish to learn. Reinforcement learning is certainly a promising approach, but we also leave this for future work.
Having excluded supervised and reinforcement learning, we move on to evolutionary-based approaches. The main advantage of this approach is that we can train directly on reducing execution times. This means we avoid the problems mentioned for supervised learning because any output resulting in the same running time reduction is rewarded equally. Using a similar genetic algorithm as Such et al. [50], we start from a population of $N$ randomly initialized GCN models. At every iteration, we assign each model a fitness score (see Section 3.2). The genetic algorithm performs truncation selection, where the top $T$ best-performing models become parents for the next generation. More specifically, when moving from one generation to the next, the best performer is copied directly to the next generation. The rest are mutated or crossovers from models among the top $T$ performers in the previous population.

### 3.1 Genetic Operators for Neural Networks

Genetic algorithms are not commonly used directly on the parameters of a model. However, there are some previous studies that have introduced genetic operators specifically for neural networks. In the following, we present the operators that we use in our training procedure.

**Mutate-Weights**

The simplest form of generating a new model is by simply copying the parent’s weights and adding Gaussian noise to them. It was the only operator used by Such et al. [50]. Montana and Davis also tested this in an earlier study [36]. However, they showed that it performs poorly compared to slightly more complicated operators.

**Mutate-Nodes**

The nodes here do not refer to the input graph’s vertices but rather the nodes of the neural network. Typically, neural networks are depicted as directed graphs, with the parameters being edge weights. Based on this picture, the *mutate-nodes* operator selects a few random non-input nodes in the neural network and adds Gaussian noise to the incoming edge weights. As for the GNN architecture presented in this paper, this operator selects random columns from the $W$ matrices to mutate, including the associated bias value. Montana and Davis [36] introduced this operator and showed it to outperform *mutate-weights*.

**Crossover-Directed**

This operator takes two parents and creates one offspring. The idea is to use the most important parameters from each parent and fill in the rest with random values [13]. More specifically, a new model is created as follows:

1. Remove a fraction $\zeta$ of the parents’ weights closest to zero.
2. Set the child’s parameters to the larger of the parents’ values.
3. Add random weights to make up for the lost values.

Faycal and Zito [13] introduced this operator and argued that the largest weights contribute the most to a model’s success and should therefore be the ones passed on to the offspring. They base this argument on another result by Mocanu et al. [35] demonstrating that neural networks with sparse connections can achieve the same accuracy as fully connected ones.
Crossover-Nodes

As in mutate-nodes, crossover-nodes refer to the nodes in the neural network, not the input graph. This operator also starts from two parents and produces one offspring as follows: For each node in the neural network – columns in the $H$ matrix – flip a coin and copy the incoming edge weights from the first or second parent [36]. The intuition for this operator is that the edge weights going into one node form a natural subgroup of all the parameters in the model. This is because these weights will determine one feature for the next layer. When combining two parents to make a new model, it is not likely that a random mix of their parameters will lead to a better-performing model. Using this crossover operator instead, learned features from each parent could combine into a better-performing offspring.

We use all the operators mentioned so far to produce the next generation. As stated earlier, the best-performing model is copied unmodified to the next generation. The rest of the population is generated by choosing a random operator and then picking one or two parents based on the selected operator. The parents are chosen uniformly at random from the top $T$ best performers from the previous generation.

3.2 Fitness Function

Deciding how to define fitness depends on what results one aims for. There are sensible aggregated measures, such as the total time to solve a set of benchmark instances or the average speedup compared to max. degree branching. The types of models one gets from training with different fitness functions can vary greatly. For example, in the case of total time, models that perform well on the longest-running instances will score the best. This model could be preferable as it may lead to more solved instances. But on the other hand, it could also perform significantly worse on easier instances. Average speedup or geometric mean of speedup are both options that treat fast and slow running instances equally. In this case, the aim is to have a model that generally makes the solver run faster. The three options mentioned so far, total, average, and geometric mean, are all reasonable. Table 1 and Figure 2 illustrates how these different fitness functions could impact the final performance.

We use the geometric mean of speedup as our fitness function since it is the most robust in Table 1 and Figure 2, and it treats all instances equally, which is reasonable when aiming for a model that performs well on unknown inputs.

<p>| Table 1 | Table of aggregated total speedup, average speedup, and the geometric mean of speedup for each of the respective fitness functions. The columns are models trained on each fitness function, and the value at each row is how the model performed according to a different aggregated measure. For this example, the baseline is max degree branching, and the solver is $BnR$. The source of these aggregated values can be seen in Figure 2. |</p>
<table>
<thead>
<tr>
<th>eval.</th>
<th>trained on</th>
<th>total</th>
<th>average</th>
<th>geom. mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>1.19</td>
<td>1.01</td>
<td>1.16</td>
<td></td>
</tr>
<tr>
<td>average</td>
<td>1.17</td>
<td>1.22</td>
<td>1.20</td>
<td></td>
</tr>
<tr>
<td>geom. mean</td>
<td>1.14</td>
<td>1.17</td>
<td>1.19</td>
<td></td>
</tr>
</tbody>
</table>
Figure 2 Illustration of how different fitness functions can lead to different results. The plot shows the speedup compared to max. degree branching for models trained with different fitness functions. Each of the models is best in their respective aggregated measure. This experiment used ten randomly selected graphs from the training data and the BnR solver. It only shows the difference fitness function can make and is otherwise unrelated to the later results.

4 Experiments

In the following, we present the computation platform, benchmark instances, details on how the GCN models were trained, and results from the experimental evaluation. When running the solvers, we record two measures: the execution time and the number of branches the solver checks. The number of branches can give an indication of how much overhead the GCN computation adds. The plots show speedup over max. degree branching, where speedup is computed by dividing the execution time of max. degree branching by the execution time of the respective alternative. Similarly, in the case of branches, the speedup is the number of branches with max. degree branching divided by the number of branches with the respective alternative. For a complete overview of the results, see the appendix.

4.1 Experimental Environment

We insert the GCN-based branching strategy into two pre-existing solvers. The first is the branch-and-reduce solver by Akiba and Iwata [1], also used by Hespe et al. [18] to find better branching strategies. And the second is the SAT-and-reduce solver by Plachetta and van der Grinten [40]. These two solvers will be referred to as BnR and SnR, respectively. Both solvers and the GCN implementation are written in C++ and compiled with g++ version 9.4.0 using the -O3 flag. The code for both of the modified solvers is publicly available on GitHub. The machine used to execute all the experiments has an AMD EPYC 7551P 32-core processor with 256GB of DDR4 ECC memory running Ubuntu 20.04.4 with Linux Kernel 5.4.0-124. We run one instance per NUMA node to speed up the experiments, so 4 instances are run concurrently. To reduce noise and maintain fairness, only a single solver runs simultaneously, and the instances are started in the same order for each solver.

2 https://github.com/KennethLangedal/CutBranching-GNN
3 https://github.com/KennethLangedal/vc-satreduce-gnn
4.2 Instances

Table 2 List of the instances used for testing. Along with the number of vertices and edges for each graph.

| Graph                  | \(|V|\)     | \(|E|\)     |
|------------------------|------------|------------|
| as-skitter             | 1,696,415  | 11,095,298 |
| baidu-relatedpages     | 415,641    | 2,374,044  |
| bay                    | 321,270    | 397,415    |
| col                    | 435,666    | 521,200    |
| ffa                    | 1,070,376  | 1,343,951  |
| hu dst ing-internallink| 1,984,484  | 14,428,382 |
| in-2004                | 1,382,870  | 13,591,473 |
| libimseti              | 220,970    | 17,233,144 |
| musae-twitch_DE        | 9,498      | 153,138    |
| musae-twitch_FR        | 6,549      | 112,666    |
| petster-fs-dog         | 426,820    | 8,543,549  |
| soc-LiveJournal1      | 4,847,571  | 42,851,237 |
| web-BerkStan           | 685,230    | 6,649,470  |
| web-Google             | 875,713    | 4,322,051  |
| web-NotreDame          | 325,730    | 1,090,108  |
| web-Stanford           | 281,903    | 1,992,636  |
| PACE, 200 graphs       | 153 - 138,141 | 625 - 227,241 |
| DIMACS, 80 graphs      | 28 - 4,000  | 72 - 3,997,732 |

We evaluate our GCN-based approach on the same instances the authors of BnR and SnR used. Both of these solvers were evaluated on instances from the PACE 2019 Challenge on Minimum Vertex Cover [11]. BnR further evaluated the branching techniques on instances from the second DIMACS Implementation Challenge [21], the Stanford Network Analysis Project (SNAP) [28], the 9th DIMACS Implementation Challenge on Shortest Paths [9], and the Network Data Repository [42]. Table 2 shows more information about the instances. Individually for each solver, the instances with less than 10 branches or less than 0.1 seconds running time are excluded for that solver. Furthermore, instances are also excluded if no branching strategy solves them within 30 minutes. When computing aggregated speedups, if one strategy times out and another finishes within 30 minutes, the strategy that times out receives the 30 minutes as execution time and the same number of branches as the strategy that finished. A time limit of 30 minutes is commonly used for evaluating exact solvers. It was used by SnR and is also the standard time limit for the PACE challenge [40, 12]. Due to the exponential running time for these solvers, it is unlikely that longer execution times would drastically change our results. BnR included results with a 10-hour time limit, and on the PACE instances, only two additional instances were solved [18].

For training the GCN models, we need to use a different dataset. The PACE 2019 Challenge drew the final 200 instances from a larger collection of over 12 thousand instances. Filtering out the 200 instances that ended up in the challenge and those with less than 0.1 seconds or more than 1 minute running time, we are left with roughly 1700 instances to use for training and validation.
4.3 GCN Training

![Figure 3](image)

**Figure 3** Results from 25 generations using the BnR solver. The shaded red area shows the range of performance from the top $T$ models on the training data. Similarly, the gray area shows the same for the validation data.

We use a population size of $N = 100$ and $T = 10$ parents for the next generation to train the GCN models. Each mutation operator is equally likely to be used, and we use $\zeta = 0.3$ for the crossover-directed operator. The models are initialized using weights drawn uniformly at random in the real interval $[-1, 1]$. The architecture of the models consists of three GCN layers with 32 features in the hidden layers. When adding noise to the weights in `mutate-weights` and `mutate-nodes`, we use a normal distribution with a mean of 0 and a standard deviation of 0.01. Finally, we remove the two highest speedups when computing the fitness. The reason for doing this is that, in some instances, the running time can be reduced to virtually zero by picking the correct vertex to branch on. When this happens, the population gets stuck since any change in that particular graph leads to a very large drop in fitness. The goal is to train a model that performs well in general, not optimally on a select few graphs.

The results from 25 generations using the BnR solver can be seen in Figure 3. The red line in the figure shows the fitness of the best model on the training data. Since we copy the best model unmodified, the fitness for this model will never decrease. However, the fitness for the rest of the models in the top $T$ can vary from generation to generation. The shaded red area in the figure illustrates this, showing the range of fitness for the top $T$ models. The black line in the figure shows the best fitness on the validation data. And similarly, the shaded black area shows the range of fitness for the top $T$ models on the validation data. Note that the best performer on the validation set is not necessarily the best on the training data.

Population-based training of neural networks like this is highly parallelizable. Evaluating the fitness of each model can be done in parallel with no communication or synchronization. However, using execution time in the fitness function made this more challenging. Modern CPUs have dynamic clock speeds that fluctuate depending on the current workload and power/temperature limitations. Loading up all the cores will typically reduce the clock speeds significantly compared to a sequential program. Both cache and memory bandwidth are also shared resources that can impact the execution time. To address these issues, we
lowered the machine’s clock speed until we got stable execution times while running several instances in parallel. Doing this introduces a new problem since it changes the cost of memory access compared to higher clock speeds. Still, the difference between running the training sequentially and in parallel is at the order of weeks compared to days, so we opt for the lowered clock speed. In the end, executing one iteration took roughly 30 minutes, with some uncertainty based on the overall quality of the particular generation. The whole training procedure took two days to finish.

### 4.4 Experimental Results for BnR

![Figure 4](image)

**Figure 4** Cactus plot showing the speedup of our GCN-based branching and targeting the packing reduction rule in the BnR solver. The basis for computing speedup is the max. degree branching strategy. Instances are sorted by the speedup over max. degree branching, individually for each configuration. Instances that timed out are ranked the lowest and marked with tle, instances that were unsolved by max. degree branching but solved by another configuration are ranked the highest and marked with new. This figure shows results for the whole set of benchmark instances (excluding those solved by no configuration and those solved quickly or with very few branches), including the speedup comparing execution time and number of branches.

We start by showing the results of our GCN-based branching strategy on the whole dataset using the BnR solver. The results are shown in Figure 4, along with the best strategy by Hespe et al. [18] targeting the packing reduction rule. The GCN-based strategy clearly outperforms both the max. degree and packing strategies. In addition to being the fastest on the majority of instances, it also solves 15 more instances that timed out with both the max. degree and packing strategies. For this solver, the speedup in terms of branches closely resembles the speedup in execution time. However, there is a more noticeable gap between the two for the GCN strategy.
The 296 graphs used for testing can be categorized as follows:

- Unsolved by any configuration: 147
- Solved with less than 10 branches or in less than 0.1 seconds: 42
- Remaining instances shown in Figure 4: 107

Regarding aggregated measures, the result is a total speedup of 2.37 over maximum degree branching, the geometric mean of the speedups is 1.94, and the average speedup is 547.04. Note that outliers heavily influence these measures. For instance, the speedup sometimes reaches the thousands on graphs where max. degree and packing timed out. Besides looking at the tables in the appendix or Figure 4, we can observe that our GCN-based branching is fastest on 78 out of 107 instances. Among the graphs with a speedup, the median speedup is 24%. In terms of checking the fewest branches, the number increases to 92 out of 107, with a median speedup of 24%.

The results are substantially better on the PACE instances than on the DIMACS and the other sparse instances. In fact, there is only one graph from the PACE challenge where our GCN-based strategy performs worse than max. degree branching. This is not surprising since the training data is more closely related to the PACE instances. In Figure 5, we show the speedup on only the DIMACS and sparse instances. It is clear that the results in terms of the number of branches are better than the execution time. This could indicate that the GCN model is choosing better branches than simply going for the highest degree, but the added overhead of invoking the GCN model outweighs the benefit. The GCN-based strategy also solves one more instance here compared to the other strategies.

4.5 Experimental Results for SnR

To the best of our knowledge, SnR is the current stat-of-the-art for exactly solving the vertex cover and independent set problems. In its default configuration, the SAT solver takes most of the execution time. With this setting, the solver looks at far fewer branches and, in extreme cases, times out while having checked virtually no branches. Since choosing these
Figure 6: Cactus plot showing the speedup of our GCN-based branching in the SnR solver. The basis for computing speedup is the max. degree branching strategy. Instances are sorted by the speedup over max. degree branching, individually for execution time and the number of branches. Instances that timed out are ranked the lowest and marked with tle. Instances that were unsolved using one strategy but solved with the other are ranked the highest and marked with new. This figure shows results for the whole set of benchmark instances (excluding those solved by no configuration and those solved quickly or with very few branches), including the speedup comparing execution time and number of branches.

branches is the only way for our GCN-based approach to make a difference, it limits the potential impact of our GCN-based branching strategy. Therefore, we also evaluate the SnR solver with reduced time for the SAT solver, later called GCN-R. In this configuration, the SAT recourse limit is halved from its default value of 8192 down to 4096.

In a similar classification as the BnR solver, the 296 graphs used for testing are distributed as follows:
- Unsolved by any configuration: 94
- Solved with less than 10 branches or in less than 0.1 seconds: 133
- Remaining instances shown in Figure 6: 69

SnR is a far stronger solver than BnR. With the default branching strategy, it solves 52 more instances than BnR with our GCN-based branching. Furthermore, the number of instances solved before any branching takes place is also significantly higher, 133 compared to 25 for BnR, highlighting the strength of the pruning techniques used by SnR.

Figure 6 shows the results of our GCN-based branching in the SnR solver compared to the default max. degree branching. Our GCN-based branching also improves this solver, but to a lesser extent. With this solver, the number of solved instances is the same for both strategies. However, they both time out on one instance where the other finished within the time limit. Regarding aggregated numbers, the average speedup is 1.08, the geometric mean of speedup is 1.06, and the total speedup is 1.03. The ratio of improved instances stays roughly the same as in BnR, with 50 out of 69 graphs having a speedup with regard to the execution time. Among the graphs with a speedup, the median speedup is 6%.
When reducing the SAT resource limit, the speedups improve. The average speedup is now 1.22, the geometric mean of speedup is 1.19, and the total speedup is 1.12. The ratio of improved instances is now much higher, at 62 out of 69 instances. Among the graphs with a speedup, the median speedup is now 21%.

5 Conclusion and Future Work

We have introduced a GCN-based branching strategy and demonstrated its effectiveness on established benchmark instances. We trained and tested our technique in two different solvers. First, we used a branch-and-reduce solver that had previously been used to find better branching strategies than simply using the highest-degree vertex. Second, we used the current state-of-the-art solver for the vertex cover and independent set problems. Our GCN-based approach gave the best branching strategy on most test instances in both solvers. For the branch-and-reduce solver, we also solved 15 more test instances.

To the best of our knowledge, this work is also the first application of evolutionary-based training of graph neural networks. We combined specialized mutation operators for neural networks to make up our training procedure, and the models were trained on unused data from the 2019 PACE Challenge. This training paradigm could open up additional avenues for machine learning on combinatorial problems. Especially where supervised or reinforcement learning is difficult to use. This is because evolutionary-based training only requires a fitness function to train the model.

As is usually the case with machine learning, more data and longer training would likely push our results further. Given more time and effort, it is likely that the other training paradigms could also be made to work for this problem – especially reinforcement learning, given its success on similar problems. There are also more types of GNN architectures to try. We have only considered the GCN model in this work. Still, other options like GIN [55], k-GNNs [37], or GraphSAGE [17] would be natural continuations.

Finally, we have only considered the independent set problem and two solvers. There are other applicable problems where we could use the technique presented in this work. For example, there were many competitive solvers that used branch-and-bound type algorithms for the directed feedback vertex set problem in the 2022 PACE challenge.

References

K. Langedal, D. Hespe, and P. Sanders


### A Results for BnR

#### Table 3 Detailed results for the BnR solver on the SPARSE instances. The s column shows speedup compared to the max. degree. The lowest execution time and number of branches appear in bold. Notice that one graph timed out with the max. degree and packing strategies. This instance receives the 30-minute running time and the same number of branches as the GCN-based strategy for the aggregated measures.

<table>
<thead>
<tr>
<th>Execution Time</th>
<th>Number of Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max degree</td>
<td>Packing s</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

#### Table 4 Detailed results for the BnR solver on the DIMACS instances. The s column shows speedup compared to the max. degree. The lowest execution time and number of branches appear in bold.

<table>
<thead>
<tr>
<th>Execution Time</th>
<th>Number of Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max degree</td>
<td>Packing s</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Execution Time</td>
<td>Number of Branches</td>
</tr>
<tr>
<td>----------------</td>
<td>------------------</td>
</tr>
<tr>
<td>Max degree</td>
<td>Packing</td>
</tr>
<tr>
<td>ve-exact</td>
<td>1.09</td>
</tr>
<tr>
<td>ve-exact</td>
<td>0.89</td>
</tr>
<tr>
<td>ve-exact</td>
<td>2.22</td>
</tr>
<tr>
<td>ve-exact</td>
<td>0.97</td>
</tr>
<tr>
<td>ve-exact</td>
<td>77.97</td>
</tr>
<tr>
<td>ve-exact</td>
<td>0.99</td>
</tr>
<tr>
<td>ve-exact</td>
<td>7.64</td>
</tr>
<tr>
<td>ve-exact</td>
<td>1.08</td>
</tr>
<tr>
<td>ve-exact</td>
<td>10.43</td>
</tr>
<tr>
<td>ve-exact</td>
<td>12.10</td>
</tr>
<tr>
<td>ve-exact</td>
<td>18.03</td>
</tr>
<tr>
<td>ve-exact</td>
<td>50.43</td>
</tr>
<tr>
<td>ve-exact</td>
<td>55.41</td>
</tr>
<tr>
<td>ve-exact</td>
<td>23.18</td>
</tr>
<tr>
<td>ve-exact</td>
<td>59.36</td>
</tr>
<tr>
<td>ve-exact</td>
<td>32.32</td>
</tr>
<tr>
<td>ve-exact</td>
<td>7.90</td>
</tr>
<tr>
<td>ve-exact</td>
<td>95.92</td>
</tr>
<tr>
<td>ve-exact</td>
<td>64.98</td>
</tr>
<tr>
<td>ve-exact</td>
<td>48.62</td>
</tr>
<tr>
<td>ve-exact</td>
<td>13.90</td>
</tr>
<tr>
<td>ve-exact</td>
<td>77.49</td>
</tr>
<tr>
<td>ve-exact</td>
<td>22.38</td>
</tr>
<tr>
<td>ve-exact</td>
<td>67.69</td>
</tr>
<tr>
<td>ve-exact</td>
<td>68.42</td>
</tr>
<tr>
<td>ve-exact</td>
<td>28.88</td>
</tr>
<tr>
<td>ve-exact</td>
<td>26.62</td>
</tr>
<tr>
<td>ve-exact</td>
<td>8.81</td>
</tr>
<tr>
<td>ve-exact</td>
<td>12.51</td>
</tr>
<tr>
<td>ve-exact</td>
<td>8.81</td>
</tr>
<tr>
<td>ve-exact</td>
<td>43.52</td>
</tr>
<tr>
<td>ve-exact</td>
<td>11.79</td>
</tr>
<tr>
<td>ve-exact</td>
<td>34.36</td>
</tr>
<tr>
<td>ve-exact</td>
<td>42.41</td>
</tr>
<tr>
<td>ve-exact</td>
<td>42.52</td>
</tr>
<tr>
<td>ve-exact</td>
<td>7.46</td>
</tr>
<tr>
<td>ve-exact</td>
<td>1,400.00</td>
</tr>
<tr>
<td>ve-exact</td>
<td>1,400.00</td>
</tr>
<tr>
<td>ve-exact</td>
<td>1,400.00</td>
</tr>
<tr>
<td>ve-exact</td>
<td>1,400.00</td>
</tr>
<tr>
<td>ve-exact</td>
<td>1,400.00</td>
</tr>
<tr>
<td>ve-exact</td>
<td>1,400.00</td>
</tr>
<tr>
<td>ve-exact</td>
<td>1,400.00</td>
</tr>
<tr>
<td>ve-exact</td>
<td>1,400.00</td>
</tr>
<tr>
<td>ve-exact</td>
<td>1,400.00</td>
</tr>
<tr>
<td>ve-exact</td>
<td>1,400.00</td>
</tr>
<tr>
<td>ve-exact</td>
<td>1,400.00</td>
</tr>
<tr>
<td>ve-exact</td>
<td>1,400.00</td>
</tr>
<tr>
<td>total</td>
<td>2,410,111</td>
</tr>
<tr>
<td>average</td>
<td>2,792,120</td>
</tr>
</tbody>
</table>
## Results for SnR

**Table 6** Detailed results for the SnR solver on the DIMACS and sparse instances. The s column shows speedup compared to the max. degree. The lowest execution time and number of branches appear in bold. Notice that some graphs timed out. These instances receives the 30-minute running time and the same number of branches as the other strategy for the aggregated measures.

<table>
<thead>
<tr>
<th>Execution time</th>
<th>Number of Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max degree</td>
<td>GCN s</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>brock200_2</td>
<td>5.87</td>
</tr>
<tr>
<td>brock200_3</td>
<td>45.25</td>
</tr>
<tr>
<td>brock200_4</td>
<td>153.50</td>
</tr>
<tr>
<td>C125.9</td>
<td>6.66</td>
</tr>
<tr>
<td>gen200_3_0.55</td>
<td>9.13</td>
</tr>
<tr>
<td>johnson16-2-4</td>
<td>3.51</td>
</tr>
<tr>
<td>keller4</td>
<td>2.96</td>
</tr>
<tr>
<td>MANN_a27</td>
<td>3.67</td>
</tr>
<tr>
<td>MANN_a45</td>
<td>240.48</td>
</tr>
<tr>
<td>p_hat300-1</td>
<td>10.78</td>
</tr>
<tr>
<td>p_hat300-2</td>
<td>98.05</td>
</tr>
<tr>
<td>p_hat500-1</td>
<td>111.63</td>
</tr>
<tr>
<td>p_hat700-1</td>
<td>1,281.83</td>
</tr>
<tr>
<td>san0000</td>
<td>1,033.05</td>
</tr>
<tr>
<td>san200_0.7_2</td>
<td>3.12</td>
</tr>
<tr>
<td>san400_0.5_1</td>
<td>28.82</td>
</tr>
<tr>
<td>san400_0.7_1</td>
<td>573.00</td>
</tr>
<tr>
<td>san200_0.7_2</td>
<td>1,311.31</td>
</tr>
<tr>
<td>san400_0.7_2</td>
<td>&gt;1,800.00</td>
</tr>
<tr>
<td>total</td>
<td>7,458.23</td>
</tr>
<tr>
<td>average</td>
<td>1.12</td>
</tr>
<tr>
<td>geom. mean</td>
<td>1.05</td>
</tr>
</tbody>
</table>
K. Langedal, D. Hespe, and P. Sanders

20:21

Table 7 Detailed results for the SnR solver on the 2019 PACE instances. The s column shows
speedup compared to the max. degree. The lowest execution time and number of branches appear
in bold. Notice that some graphs timed out. These instances receives the 30-minute running time
and the same number of branches as the other strategy for the aggregated measures.

vc-exact_005
vc-exact_006
vc-exact_010
vc-exact_019
vc-exact_031
vc-exact_035
vc-exact_037
vc-exact_038
vc-exact_041
vc-exact_042
vc-exact_043
vc-exact_044
vc-exact_045
vc-exact_046
vc-exact_047
vc-exact_048
vc-exact_049
vc-exact_050
vc-exact_051
vc-exact_052
vc-exact_053
vc-exact_054
vc-exact_055
vc-exact_056
vc-exact_057
vc-exact_058
vc-exact_059
vc-exact_060
vc-exact_061
vc-exact_062
vc-exact_063
vc-exact_064
vc-exact_065
vc-exact_066
vc-exact_067
vc-exact_068
vc-exact_069
vc-exact_070
vc-exact_071
vc-exact_072
vc-exact_073
vc-exact_074
vc-exact_077
vc-exact_081
vc-exact_082
vc-exact_083
vc-exact_091
vc-exact_093
vc-exact_200
total
average
geom. mean

Max degree
14.24
4.67
11.26
12.81
262.48
47.16
65.20
536.75
283.35
209.96
682.69
830.47
141.62
479.52
432.25
172.49
143.05
289.93
368.18
112.94
367.94
203.35
42.68
484.93
533.33
296.89
94.05
441.34
106.38
339.35
364.76
203.36
214.95
65.16
746.68
83.35
244.19
74.72
199.15
307.34
287.55
48.24
94.11
910.11
356.12
1,114.70
1,156.70
684.89
1,384.23
16,551.54

Execution time
GCN s
GCN-R
1.38
10.31
9.26
1.00
4.66
3.99
1.19
9.43
8.71
1.06
12.13
11.52
1.11
236.28
228.99
1.01
46.91
40.29
1.14
57.22
49.67
1.06
504.55
336.93
1.14
249.62
228.26
1.05
200.84
176.79
575.96
1.19
608.90
1.11
747.88
633.91
1.08
130.87
117.05
1.14
420.71
402.01
1.07
403.83
338.20
0.92
187.00
161.74
0.90
158.20
119.71
1.21
240.01
214.05
1.05
351.44
299.89
0.86
131.90
99.80
1.00
367.61
325.34
1.12
181.03
160.81
0.82
52.20
41.26
1.01
481.31
400.77
0.95
564.06
382.75
1.00
296.55
233.02
1.05
89.48
73.01
1.00
441.46
378.06
1.03
103.28
92.24
0.90
377.05
324.93
1.03
354.49
308.13
1.07
190.20
164.95
1.07
201.38
176.85
1.15
56.59
46.15
1.07
695.39
599.71
39.53
2.11
43.90
1.15
213.16
186.07
1.13
66.14
54.57
1.21
164.80
129.95
0.88
349.41
262.03
1.01
285.56
254.10
1.01
47.57
38.21
1.05
89.49
73.04
1.06
855.98
717.73
1.11
321.09
292.62
0.88
1,267.89
962.21
0.98
1,184.84
977.77
0.92
743.10
600.95
>1,800.00 <0.77
>1,800.00
1.00 14,190.79
16,560.38
1.07
1.05

s
1.54
1.17
1.29
1.11
1.15
1.17
1.31
1.59
1.24
1.19
1.12
1.31
1.21
1.19
1.28
1.07
1.19
1.35
1.23
1.13
1.13
1.26
1.03
1.21
1.39
1.27
1.29
1.17
1.15
1.04
1.18
1.23
1.22
1.41
1.25
1.90
1.31
1.37
1.53
1.17
1.13
1.26
1.29
1.27
1.22
1.16
1.18
1.14
<0.77
1.17
1.24
1.23

Max degree
21,373
10,321
24,805
25,573
751,333
103,415
139,951
47,564
560,171
412,043
1,987,935
1,169,595
223,855
1,385,345
700,511
277,717
265,311
524,847
605,749
222,787
742,883
396,235
77,763
605,493
369,193
389,189
153,767
750,693
237,715
580,635
539,377
379,593
330,061
129,915
718,307
74,777
431,177
141,555
297,773
383,985
526,765
85,133
153,767
1,129,433
868,967
1,692,351
1,959,583
1,035,161
163
24,641,610

Number of Branches
GCN s
GCN-R
16,341
1.31
16,337
8,743
1.18
8,897
21,525
1.15
21,791
21,141
1.21
21,141
618,845
1.21
624,627
92,773
1.11
92,965
115,079
1.22
118,101
13,571
3.50
15,224
446,575
1.25
448,417
318,093
1.30
323,581
1,510,951
1.32
1,706,531
927,837
1.26
958,159
184,035
1.22
184,411
1,078,477
1.28
1,095,625
532,225
1.32
554,569
265,105
1.05
269,873
225,279
1.18
231,625
411,621
1.28
420,361
479,097
1.26
490,917
179,035
1.24
187,557
590,957
1.26
608,933
314,093
1.26
317,659
67,797
1.15
69,833
443,437
1.37
461,611
267,187
1.38
312,279
328,549
1.18
350,827
133,153
1.15
134,747
545,615
1.38
555,759
207,215
1.15
209,261
477,137
1.22
495,085
446,631
1.21
539,917
318,951
1.19
323,193
263,297
1.25
274,535
105,033
1.24
106,829
580,471
1.24
622,477
61,695
1.21
76,097
332,151
1.30
339,629
122,765
1.15
122,913
193,943
1.54
202,583
289,101
1.33
299,845
421,175
1.25
433,831
83,435
1.02
84,653
133,153
1.15
134,747
817,385
1.38
876,193
710,671
1.22
722,971
1,121,237
1.51
1,204,803
1,523,743
1.29
1,684,477
830,441
1.25
883,663
>163 <1.00
>163
19,196,929
1.28 20,240,222
1.29
1.26

s
1.31
1.16
1.14
1.21
1.20
1.11
1.19
3.12
1.25
1.27
1.16
1.22
1.21
1.26
1.26
1.03
1.15
1.25
1.23
1.19
1.22
1.25
1.11
1.31
1.18
1.11
1.14
1.35
1.14
1.17
1.00
1.17
1.20
1.22
1.15
0.98
1.27
1.15
1.47
1.28
1.21
1.01
1.14
1.29
1.20
1.40
1.16
1.17
<1.00
1.22
1.23
1.21

SEA 2024


Abstract

Rank and select data structures seek to preprocess a bit vector to quickly answer two kinds of queries: \textsc{Rank}(i) gives the number of 1 bits in slots 0 through i, and \textsc{Select}(j) gives the first slot s with \textsc{Rank}(s) = j. A succinct data structure can answer these queries while using space much smaller than the size of the original bit vector.

State of the art succinct rank and select data structures use as little as 4\% extra space (over the underlying bit vector) while answering rank and select queries very quickly. Rank queries can be answered using only a handful of array accesses. Select queries can be answered by starting with similar array accesses, followed by a linear scan through the bit vector.

Nonetheless, a tradeoff remains: data structures that use under 4\% space are significantly slower at answering rank and select queries than less-space-efficient data structures (using, say, over 20\% extra space).

In this paper we make significantly progress towards closing this gap. We give a new data structure, SPIDER, which uses 3.82\% extra space. SPIDER gives the best known rank query time for data sets of 8 billion or more bits, even compared to much less space-efficient data structures. For select queries, SPIDER outperforms all data structures that use less than 4\% space, and significantly closes the gap in select performance between data structures with less than 4\% space, and those that use more (over 20\% for both rank and select) space.

SPIDER makes two main technical contributions. For rank queries, it improves performance by interleaving the metadata with the bit vector to improve cache efficiency. For select queries, it uses predictions to almost eliminate the cost of the linear scan. These predictions are inspired by recent results on data structures with machine-learned predictions, adapted to the succinct data structure setting. Our results hold on both real and synthetic data, showing that these predictions are effective in practice.

2012 ACM Subject Classification  Theory of computation \textemdash\ Sorting and searching

Keywords and phrases  Rank and Select, Succinct Data Structures, Data Structures, Cache Performance, Predictions

Digital Object Identifier  10.4230/LIPIcs.SEA.2024.21

Supplementary Material  Software: https://github.com/williams-cs/spider
archived at swh:1:dir:5e78bf17872ee5fcccfebab77a6e238045b8dc199

Funding This research was supported in part by NSF grant CCF 2103813.
Introduction

Rank and select are classic bit vector queries in computer science. Given a bit vector $V$, $\text{RANK}(i)$ gives the number of 1 bits among $V[0], \ldots, V[i]$. $\text{SELECT}(j)$ gives the location of the $j$th 1 bit; in other words, $\text{SELECT}(j)$ gives the smallest $i$ such that $\text{RANK}(i) = j$. Thus, if $n$ is the length of $V$, and $n_1$ is the number of 1 bits it contains, rank queries are well-defined for $i = 0, \ldots, n - 1$ and select queries are well-defined for $i = 1, \ldots, n_1$.

Rank and select data structures are most interesting when they are restricted to small space. After all, with unlimited space one could just store the answer to all $n + n_1$ rank and select queries. Thus, we focus on succinct data structures: those that use much less than $n$ bits of space in addition to $V$.

Rank and select queries can be used to implement many other succinct data structures, including, for example, trees and graphs [17, 25], suffix trees [14], and filters [22, 13]. Wavelet trees are a particularly motivating example, as they use a sequence of rank and select data structures as a subroutine [18, 8], and have applications in areas such as text compression, DNA alignment, and computational geometry [4]. We note that all of these applications require a data structure that can handle both rank and select queries on $V$.

These applications have motivated a long line of work on engineering rank and select data structures that can answer queries quickly with very small space overhead [28, 27, 9, 16, 7, 20]. When space is at a premium, the state of the art result is pasta-flat by Kurpicz [11]. His data structure uses only 3.58% extra space ($0.0358n$ bits in addition to $|V|$), and is very performant, achieving query time up to 16.5% faster than the previous state of the art.

However, rank and select query performance still incurs a tradeoff between query time and space. For example, Vigna [27] gives a data structure for rank queries using 25% space that is roughly 19% faster than pasta-flat, and a data structure for select queries using 12.2% space, which is roughly 65% faster than pasta-flat.

In this paper, we make significant progress on overcoming this tradeoff.

1.1 How Rank and Select Data Structures Work

We give a high-level description of the basic ideas behind rank and select data structures. This is a useful introduction to our data structure, and gives context to related work. We begin with the naive data structure: one can store the solution to all rank and select queries, requiring $64(n + n_1)$ bits of space. Throughout the paper we give data structures that allow for $n$ up to $2^{64}$, and thus each bit vector index requires 64 bits.

1.1.1 Searching in a Cache Line

Rank and select data structures often operate on bit vectors with billions of bits. Therefore, cache efficiency is by far the most important consideration for performance. In this paper we assume a cache line size of 512 bits and a word size of 64 bits. (Our results likely generalize by adjusting the parameters in Section 2.)

With cache efficiency in mind, we can improve the naive rank and select data structure. To begin, we observe that there is no need to store metadata to help with search within a cache line. This is because cache efficiency is our primary goal: a single cache miss allows us to bring in 512 bits, and look for 1s manually. First we describe how this can be used to save space, and second we describe how to decrease the computation cost within a cache line. This is the strawman data structure, also described in [28].
Saving Space. The strawman data structure has a rank array of length \([n/512]\) that stores the number of 1s before each cache line: its \(i\)th entry stores \(\text{RANK}(512i−1)\). To answer \(\text{RANK}(i)\), first look up the \([i/512]\)th entry in the rank array and store its value in \(r_1\) (thus, \(r_1\) is the number of 1 bits in slots 0 through \(512\lfloor i/512 \rfloor −1\) of \(V\)). Then, access \(V\) to find the number of 1s in slots \(512\lfloor i/512 \rfloor \) through \(i\); adding this to \(r_1\) gives \(\text{RANK}(i)\).

For select, the strawman data structure uses a heuristic common to most succinct data structures, called rank-based select [11]. First, set a parameter \(\sigma\) (often \(\sigma = 8192\)). Similar to rank, the strawman data structure stores the select array, whose \(i\)th slot stores \(\text{SELECT}(\sigma \cdot i + 1)\), for \(i = 0, \ldots, \lfloor (n_1 - 1)/\sigma \rfloor\).

The idea of rank-based select is to use the select array to get an approximate solution, and then refine it with a linear search — but we note that using the rank array saves considerable time for this search. To find \(\text{SELECT}(i)\), use the select array to look up \(s = \text{SELECT}(\lfloor (i - 1)/\sigma \rfloor + 1)\). Then, begin a linear scan in the rank array from position \([s/512]\); the goal is to find the largest entry in the rank array that is smaller than \(i\). Let \(e\) be this entry and \(p\) be its position in the rank array; once \(e\) and \(p\) are found, search for the \((i - e)\)th 1 bit within the cache line starting at \(512p\) in \(V\).

We note that rank-based select is \(\Omega(n)\) time in the worst-case due to the linear scan, but past work has shown that it is efficient in practice [11, 28].

The strawman data structure uses, for \(\sigma = 8192, 64 \cdot n/512 + 64 \cdot n_1/8192 \leq .133n\) space (13.3% overhead), and is very cache-efficient. However, the cost for a naive scan within a cache line is considerable. Fortunately, as we now explain, many modern machines support low-level operations that make this cost almost negligible.

Rank Search Within a Cache Line. Many modern processors have a popcount operation that gives the number of 1s in a 64-bit word. Thus, we can find the number of 1s in a cache line at or before position \(i\) with at most 8 popcount operations. Before the final popcount operation, we must right shift to remove all bits in the word after \(i\). This results in an extremely computationally efficient way to compute the rank within a cache line.

Select Search Within a Cache Line. Pandey, Bender, and Johnson [23] give a method to search within a bit vector using a clever application of specialized processor instructions. In particular, many modern x86 machines support the pdep and tzcnt instructions. A single application of these two instructions allows us to find the location of the \(i\)th 1 bit in a 64-bit word; see [23] for details. Thus, we can find the \(j\)th 1 bit within a cache line as follows. We perform up to 8 popcount instructions, keeping a running count \(c\) of the number of 1s seen. If adding the popcount of the new word would cause \(c > j\), we instead use pdep and tzcnt to find the \((j - c)\)th 1 bit within the word.

This methodology is often called fast select, and it noticeably speeds up select performance on the machine we used in our experiments (see, for example, the difference between pasta-flat and pasta-flat-fs in Figure 3). We note that, except for pasta-flat which we keep as a baseline, all data structures we compare to in our select experiments use fast select within a cache line.

1.1.2 Multi-Level Rank Data Structures

To reduce space past 13%, many data structures use a multi-level rank data structure.

Consider splitting the bit vector into blocks of size \(2^b\) (we will retain a linear search in the cache line, so assume \(2^b \gg 512\)). Then, we can store two rank arrays. The \(i\)th entry in the high-level rank array stores the number of ones before bit \(2^b \cdot i\) in \(V\). The low-level rank array recurses within a \(2^b\)-size block: the \(j\)th entry of the low-level rank array stores the number of ones between bit \(2^b \cdot \lfloor 512 \cdot j/2^b \rfloor\) and bit \(512j\) of \(V\).
To find \( \text{Rank}(i) \) we sum the \( \lfloor i/2^b \rfloor \)th entry in the high-level array, the \( \lfloor i/512 \rfloor \)th entry in the low-level array, and finally a scan within the cache line beginning at slot \( 512 \lfloor i/512 \rfloor \) of \( V \). Select searches can be handled similarly: first a linear scan is performed in the high-level rank array, and then in the low-level rank array, finally searching within a cache line.

The multi-level approach incurs an extra array lookup in exchange for less space. The top-level array requires space \( 64 \cdot n/2^b \). Each entry in the second-level array is \( < 2^b \) and can therefore be stored in \( b \) bits, giving space \( b \cdot n/512 \). The most performant and space-efficient data structures recurse another time, using three rank arrays [28, 11].

### Improving Select Queries with Multi-Level Rank.

The multi-level rank data structure improves select queries as well. For example, consider a select query where \( \text{Select}(\sigma \lfloor i/\sigma \rfloor) \) is \( 10 \cdot 2^b \) slots away from \( \text{Select}(i) \). With a single rank array, the algorithm would have to scan through roughly \( 10 \cdot 2^b/512 \) array slots. Meanwhile, with the two-level data structure, the algorithm would scan through roughly 10 entries in the top-level array, followed by \( 2^b/512 \) entries in the low-level array; for large \( b \) nearly a factor 10 speedup.

This speedup leads to a tension between rank and select queries. Optimizations to the rank data structure are limited by their impact on select performance. This tradeoff can be seen in the pasta-wide data structure of Kurpicz [11]. pasta-wide is a simple two-level rank data structure with extremely strong performance; unfortunately, the simplified rank array makes select queries extremely slow. This holds even if the select queries use binary search in an attempt to speed up performance.

One of the main contributions of SPIDER is to resolve this tension: we show how predictions can reduce the cost of a select query even for highly efficient rank data structures, achieving the best of both worlds.

### Interleaving Rank Arrays.

For sufficiently large bit vectors, accessing each rank array may incur a separate cache miss, significantly impacting performance. For this reason, Zhou et al. [28] introduced the idea of interleaving the rank arrays. Specifically, in the above description, we access entry \( k \) of the high-level rank array only when querying \( \text{Rank}(i) \) with \( k2^b \leq i < (k + 1)2^b \); thus the entry \( j \) accessed in the low-level rank array satisfies \( j \in \{k2^b/512, \ldots, (k + 1)2^b/512 - 1\} \). Thus, Zhou et al. store a single array storing (for each \( k \)) the \( k \)th entry of the high-level array, followed by all possible lower-array entries given above. If these entries all fit in 512 bits, both arrays can be accessed with a single cache miss.

### 1.2 Our Contributions

We give a data structure, SPIDER (a Succinct Predictive InDEX for Rank and select) which improves rank and select performance.

For rank queries, SPIDER uses a simple two-level data structure – but interleaves the lower-level rank array with the bit vector itself. Thus, if the high-level rank array fits in cache, SPIDER can answer rank queries with a single cache miss. This noticeably improves rank query time, even compared to data structures that use far more space.

For select queries, SPIDER avoids a costly scan using predictions. In particular, we draw inspiration from recent work on learned data structures, e.g. [10, 5]. Learned data structures use best-fit lines to predict the answer to queries; past work has shown that real world data is roughly piecewise linear, leading to good results. SPIDER applies this principle to select queries. We assume that successive select array entries are separated by approximately-evenly-spaced 1s, and use this assumption to warm-start our linear scan.
Table 1 Comparing the performance of succinct rank and select data structures. See Section 3 for details of the experiments. We list the space used as a percentage of the size of the original bit vector, and the average time for rank and select on a 8 billion bit segment of the Wikipedia dataset. The best performance numbers are in bold, and the best among data structures with under 5% space are italicized.

<table>
<thead>
<tr>
<th>Structure</th>
<th>Cite</th>
<th>Space</th>
<th>Rank(ns)</th>
<th>Select(ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>vigna-rank</td>
<td>[27]</td>
<td>25.00</td>
<td>40.69</td>
<td>179.15</td>
</tr>
<tr>
<td>sdsl-v</td>
<td>[6]</td>
<td>25.00</td>
<td>40.31</td>
<td>-</td>
</tr>
<tr>
<td>pasta-wide</td>
<td>[11]</td>
<td>3.23</td>
<td>38.81</td>
<td>-</td>
</tr>
<tr>
<td>vigna-select</td>
<td>[27]</td>
<td>12.20</td>
<td>-</td>
<td>95.04</td>
</tr>
<tr>
<td>vigna-select-H</td>
<td>[27]</td>
<td>15.63</td>
<td>-</td>
<td>71.56</td>
</tr>
<tr>
<td>ni-spider</td>
<td>§2.3</td>
<td>3.62</td>
<td>36.98</td>
<td>138.61</td>
</tr>
<tr>
<td>spider</td>
<td>§2</td>
<td>3.83</td>
<td><strong>33.56</strong></td>
<td>126.96</td>
</tr>
</tbody>
</table>

SPIDER also uses the multi-level strategy on the select array to allow us to sample more bits, further improving select performance; to our knowledge this is the first data structure to use a multi-level select array.

We also give a variant, Non-Interleaved SPIDER, which does not interleave the rank array with the bit vector. Non-Interleaved SPIDER’s rank performance is worse than SPIDER but competitive with the state of the art; its select performance is similar to SPIDER.

Overall, SPIDER gives improved bounds over any known succinct data structure for rank queries. For select queries, SPIDER improves on the state of the art data structure with less than 5% extra space usage, and nearly matches the performance of the best known succinct data structure that uses 12.5% space and cannot answer rank queries. In particular, compared to the best known data structure with under 5% space, SPIDER gives up to a 22% speedup for rank, and 41% speedup for select.

1.3 State of the Art Rank and Select

We use the following rank and select data structures as a baseline to compare our data structure. See Table 1 for a summary of the advantages of each.

**poppy**: Uses a three-level strategy, interleaving the last two levels, with carefully implemented bit tricks to improve rank and select performance [28].

The publicly-available implementation contains a bug that results in incorrect queries for bit arrays beyond 4 billion bits.

**pasta-flat**: Introduced by Kurpicz [11], pasta-flat achieves the state of the art performance for rank and select data structures with less than 5% space overhead. The basic idea of pasta-flat is a three level data structure with interleaved second and third levels, much like poppy. However, pasta-flat uses a different strategy for interleaving the bottom two arrays using 128-bit entries. These entries are handled efficiently using SIMD operations. Overall, Kurpicz showed that pasta-flat gave a significant speedup over poppy (therefore, we only compare to pasta-flat in our tests).

One advantage of pasta-flat is that it supports both select_1 queries (as defined in this paper), as well as select_0 queries, which give the location of the jth 0. We do not consider select_0 queries; however, SPIDER can be easily extended to handle select_0 queries by using a second select data structure (that stores the location of sampled 0s rather than sampled 1s), slightly increasing the space.
pasta-flat-fs: Identical to pasta-flat, but uses fast select [23] within each cache line.

pasta-wide: The second pasta data structure [11] uses a simple two-level structure for very fast rank queries. Adding the select structure from pasta-flat results in large space and slow select performance; therefore we only consider rank queries on pasta-wide.

vigna-rank: Vigna’s rank structure, rank9, follows a two level approach [27]. Entries in the two levels are grouped into 128-bit “words” that are unpacked during select queries using broadword programming techniques. Rank9 has a select data structure, select9, but we do not include it in Section 3 because it is much slower than vigna-select.

vigna-select: Vigna also proposes a select-only structure (called “simple select” in [27]) which we refer to as vigna-select. vigna-select is a position based select structure (it is not rank based); see [27] for more details. This data structure takes a tuning parameter; we use 2 as it gave the best performance (see also the comparison in [11]).

vigna-select-H: uses similar techniques to vigna-select but is specifically optimized for bit vectors where each bit is a 1 with probability 5.

sdsl-v: The Succinct Data Structure Library [6] contains an implementation of the vigna-rank data structure described above.

1.4 Other Related Work

There are tight upper and lower theoretical bounds for rank and select data structures [24, 12].

Compressed Bit Vectors. An orthogonal line of work has looked at how to save space by compressing the underlying bit vector $V$. In particular, many practical bit vectors can be stored using much less than $n$ bits of space. The goal of this line of work is a trade-off between how much space is used on practical bit vectors, and how much time rank and select queries take. See for example [25, 21, 19, 2].

There are two downsides to compressing $V$. First, this compression is situational: some bit vectors have high entropy and cannot be compressed. Second, queries require uncompressing the underlying data, which results in a cost overhead; see e.g. the experiments in [2].

In this paper we assume that $V$ is stored uncompressed. It is plausible that our ideas could be combined with past work to speed up compressed queries as well.

Learned Data Structures. A recent, exciting line of work has looked at learned data structures [10, 15, 5]. Rather than keeping a worst case index, learned data structures use machine learning techniques to store high-level information about the data.

More recently, Boffa, Ferragina, and Vinciguerra [2] used a learned index to achieve an improved data structure for rank and select. Their focus was distinct from ours: their goal was to compress the bit vector itself (as mentioned above), whereas our goal is to store succinct metadata to speed up queries.

However, SPIDER’s predictive method for select uses a structure that is reminiscent of these learned indices. In particular, the basic strategy of learned indices is to store a sequence of best-fit lines on the underlying data. Past results indicate that on real world data, best fit lines can often store much more accurate and much more concise information than classic worst-case data structures [10, 15, 3, 5].

One can view our select methodology as a very lightweight application of these learned indices. We do not have space to explicitly store best-fit lines; instead, we look up the number of ones in a subarray to roughly estimate what a best fit line could look like for a given subsection of the bit vector. This allows us to estimate the best place to start when scanning the bit vector during a select query.
Table 2 Table of notation used in this paper.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V$</td>
<td>The bit vector we are querying.</td>
</tr>
<tr>
<td>$n$</td>
<td>Length of the bit vector.</td>
</tr>
<tr>
<td>$n_1$</td>
<td>Number of 1 bits in the bit vector.</td>
</tr>
<tr>
<td>$i$</td>
<td>A given query (Rank or Select).</td>
</tr>
<tr>
<td>$s$</td>
<td>The predicted superblock index.</td>
</tr>
<tr>
<td>$r_1$</td>
<td>Number of 1 bits in all slots before the superblock containing the query.</td>
</tr>
<tr>
<td>$b_2$</td>
<td>Index of the basic block containing the query.</td>
</tr>
<tr>
<td>$r_2$</td>
<td>16 bit count prepended to the original block; represents the number of 1 bits from the start of the superblock until the start of the basic block.</td>
</tr>
<tr>
<td>$\sigma_h$</td>
<td>High-level sampling threshold: the frequency with which locations are stored in the high-level select array.</td>
</tr>
<tr>
<td>$\sigma_r$</td>
<td>Low-level sampling threshold: the frequency with which locations are stored in the low-level select array.</td>
</tr>
<tr>
<td>$\ell$</td>
<td>Index in the low-level select array; gives a lower bound on the query solution.</td>
</tr>
<tr>
<td>$p$</td>
<td>Prediction for a select query.</td>
</tr>
<tr>
<td>$p'$</td>
<td>Altered prediction when the sampled bits cross a superblock boundary.</td>
</tr>
<tr>
<td>$B$</td>
<td>First basic block we search in given $p$ or $p'$.</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>Sampling threshold for Non-Interleaved SPIDER.</td>
</tr>
</tbody>
</table>

2 The SPIDER Data Structure

We now describe our data structures: first SPIDER, and then Non-Interleaved SPIDER. Then we compare the techniques of SPIDER and Non-Interleaved SPIDER and how these differences impact performance.

2.1 SPIDER Rank

SPIDER Rank Data Structure. SPIDER stores two arrays to help answer rank queries: a rank array, and a modified bit vector.

First, consider partitioning the bit vector $V$ into $n/63488$ contiguous superblocks of 63488 bits\(^1\) (we explain this constant below). Thus, we say that superblock $i$ contains a slot $s$ if $63488i \leq s < 63488(i + 1)$. The rank array stores the number of 1s before the first bit in each superblock.\(^2\) Thus, the rank array consists of $n/63488$ 64-bit entries and the $i$th entry in the rank array stores $\text{Rank}(63488\lfloor 496i/63488 \rfloor - 1)$.

Then, we store a modified bit vector. We partition $V$ into $n/496$ contiguous original blocks of 496 bits. Thus, each superblock consists of 128 original blocks. The modified bit vector consists of $n/496$ basic blocks, each of 512 bits. For original block $i$ starting at slot $s_o$, let $s_b$ be the first slot in the superblock containing $s_o$; we define the local rank to be the number of 1s in slots $s_b$ through $s_o - 1$ (inclusive). Thus, the local rank of original block $i$ is $\text{Rank}(496i - 1) - \text{Rank}(63488\lfloor 496i/63488 \rfloor - 1)$. The $i$th basic block has two parts: first, a 16-bit number storing the local rank of the $i$th original block, followed by the 496 bits of the $i$th original block. The modified bit vector consists of all basic blocks.

---

\(^1\) We assume for simplicity that the bit vector is padded to have length a multiple of 63488.

\(^2\) The first entry in the rank array is always 0, but we store it anyway for simplicity. Recall that $\text{Rank}(i)$ is defined to be inclusive of $i$ – thus, the number of bits before slot $i$ is $\text{Rank}(i - 1)$. 
After the modified bit vector has been constructed, we no longer need to store \( V \). (Its contents are present in the modified bit vector anyway.) Thus, our data structure consists only of the rank array and the modified bit vector.

**Preprocessing.** The rank array and the modified bit vector can be created simultaneously during a linear scan over the original bit vector.

**Queries.** On a query \( \text{Rank}(i) \), we begin by using the rank array to find \( r_1 = \text{Rank}(63488 \lfloor i/63488 \rfloor - 1) \) (this can be done by looking up the \( \lfloor i/63488 \rfloor \)th position in the rank array).

Then, we access the modified bit vector. Bit \( i \) can be found in the \( b_2 = \lfloor i/496 \rfloor \)th basic block. Let \( r_2 \) be the 16-bit value stored at the beginning of block \( b_2 \) in the modified bit vector. Then \( r_2 + r_1 \) gives the rank of the last slot before \( b_2 \).

Finally, we must count the number of 1s before position \( i \) within \( b_2 \). This can be found using the \texttt{popcount} instruction on the first \( \lfloor i/64 \rfloor + 1 \) words of the 8 words in the basic block. During this process, we must mask out the first 16 bits of the first word, and then remove all bits after \( i \) in the last word using a shift operation. Summing the result of all of these \texttt{popcount} operations with \( r_2 + r_1 \) gives the rank of \( i \).

**How we Chose the Superblock Size.** The key parameter we chose is the size of the local rank; we chose a moderate value of 16 bits. If the local rank fits in 16 bits, each superblock must have at most \( 2^{16} \) slots. The basic block is the size of a cache line (512 bits), and 16 bits are used to store the local rank, leaving 496 bits to store the original basic block.

The size of the superblock is chosen to be a power of 2 times the size of a basic block. When calculating the rank, we divide \( i \) by first the size of a superblock, and later divide \( i \) by the size of a basic block. This requires two divisions, a potentially-expensive CPU operation. We reduce this to one division and one shift by setting the size of a superblock to 128 times the size of a basic block: thus, after we calculate \( \lfloor i/63488 \rfloor \), we can right shift by 7 to get \( \lfloor i/496 \rfloor \). This choice slightly increases our space, but significantly reduces query time.

### 2.2 SPIDER Select

Now we describe how SPIDER works on \texttt{Select()} queries. We use a rank-based approach, so select queries also access the rank array and modified bit vector to help guide the query.

**Storing Metadata.** We store two arrays in addition to the rank array and modified bit vector: a high-level select array, and a low-level select array. In short, we augment the rank-based select strategy described in Section 1.1.1 with the multi-level strategy.

We define two sampling thresholds, one for each select array: a high-level sampling threshold \( \sigma_h \), and a low-level sampling threshold \( \sigma_l \).

First, we store a **high-level select array** of length at most \( n/63488 + 2 \). We define the high-level sampling threshold to be \( \sigma_h = 2^\lceil \log_2(63488 \cdot n_1/n) \rceil \). We will see that the high-level select array has \( 2 + \lfloor n_1/\sigma_h \rfloor \) entries; thus, \( \sigma_h \) is the smallest power of 2 that ensures that the high-level select array has at most \( n/63488 + 2 \) entries.\(^3\)

The idea of the high-level select array is to store, for every \( \sigma_h \)th 1 bit, the nearest superblock entry of that 1 bit. The nearest superblock boundary to slot \( s' \) is given by \( \lfloor s'/63488 + 1/2 \rfloor \). Thus, entry 0 in the high-level select array stores \( \lceil \text{Select}(1)/63488 + \)

---

\(^3\) Choosing \( \sigma_h \) to be a power of 2 allows us to replace a division with a shift, improving query time.
1/2]. Then, entry $i$ in the high-level select array (for $i$ from 1 to $\lceil (n_1 - 1)/\sigma_h \rceil$) stores $\text{SELECT}(i \cdot \sigma_h + 1)/63488 + 1/2$. Finally, we store $n/63488 - 1$ in the last element of the high-level select array. Thus, the high-level select array has $\lceil (n_1 - 1)/\sigma_h \rceil + 2$ entries, each of 64 bits.

Then, we store a low-level select array. Our goal is to have each entry in the low-level select array take only 16 bits – thus, drawing inspiration from the local rank used in Section 2.1, we define the superblock offset of any slot $s'$ to be the number of slots between $s'$ and the first slot of the superblock containing $s'$. The first slot of the superblock containing $s'$ is $63488 \cdot s'/63488$, so the superblock offset of $s'$ is $s' - 63488 \cdot s'/63488$.

We define the low-level sampling threshold $\sigma_l = 2^{\lfloor \log_2(4096 \cdot 0.99n_1/n) \rfloor}$. The low-level select array has $2 + \lceil n_1/\sigma_l \rceil$ entries, each storing a superblock offset. Thus, $\sigma_l$ is the smallest power of 2 such that the low-level select array stores at most $2 + n/(0.99 \cdot 4096)$ entries.\footnote{The .99 term is to help performance for bit vectors where very slightly over half of the bits are 1s: without this term, if (say) $n_1/n = .50001$, we would store roughly $n/8192$ entries (whereas we would store $n/4096$ for $n_1/n = .5$), hurting select query performance. Thus, the .99 term slightly increases our worst-case space bound, but helps performance on datasets with $n_1/n > .5$ and $n_1/n \leq .505$.}

The $i$th entry in the low-level select array stores the superblock offset of $\text{SELECT}(\sigma_l \cdot i)$. In other words, the first slot of the superblock containing $\text{SELECT}(\sigma_l \cdot i)$ is slot $s_i = 63488 \cdot \text{SELECT}(\sigma_l \cdot i)/63488$; thus the $i$th entry stores $\text{SELECT}(\sigma_l \cdot i) - s_i$. The last entry of the low level select array stores $\text{SELECT}(n_1) - 63488 \cdot \text{SELECT}(n_1)/63488$. Each entry has value at most 63488, so each low-level select array entry can be stored in 16 bits.

Preprocessing. First we build the rank array and modified bit vector in one scan. Then we can calculate $n_1 = \text{RANK}(n - 1)$, from which we obtain $\sigma_h$ and $\sigma_l$. Then, in a second scan through the data we build the high level select array and low level select array.

Queries and Predictions. On a query $\text{SELECT}(i)$, we use the high-level select array to find the superblock containing the $i$th set bit as follows. We first set $s$ to be entry $\lceil (i - 1)/\sigma_h \rceil$ in the high-level select array; $s$ is a guess for the superblock containing $\text{SELECT}(i)$.

We then do a linear scan starting at $s$ to find the superblock containing $\text{SELECT}(i)$. We begin with the values stored in the $s$th and $(s + 1)$st entries in the rank array; if $i$ is between these values then $\text{SELECT}(i)$ is in the $s$th superblock. Otherwise, we begin a linear scan for the correct superblock: if $i$ is less than the $s$th rank array entry we decrement $s$ and recurse; if $i$ is greater than the $(s + 1)$st rank array entry we increment $s$ and recurse. At this point, we know that $\text{SELECT}(i)$ is in the $s$th superblock.

Now we look in the low-level select array. Let $a$ be the value stored in the $b$th entry of the low-level select array with $\ell = \lceil (i - 1)/\sigma_l \rceil$; let $b$ be the value stored in the $(\ell + 1)$st entry of the low-level select array.

We now create our prediction for $\text{SELECT}(i)$. First, some motivation. We say that $i$ well-specified if $a$ and $b$ both represent slots in the $s$th superblock – that is to say, $i$, is well-specified if the $s$th superblock contains $\text{SELECT}(\sigma_l \cdot \ell)$ and $\text{SELECT}(\sigma_l \cdot (\ell + 1))$. See Figure 1a. If $i$ is well-specified, $\text{SELECT}(i)$ must be between $63488 \cdot s + a$ and $63488 \cdot s + b$.

Let’s assume (momentarily) that $i$ is well-specified, and further, that all 1 bits are spread evenly between slots $63488 \cdot s + a$ and $63488 \cdot s + b$. There are $\sigma_l$ 1s between these slots by definition, so if they are spread evenly, there is a 1 every $(b - a)/\sigma_l$ slots. By definition, there are $i - \sigma_l \cdot \ell$ 1s between $\text{SELECT}(\sigma_l \cdot \ell)$ and $\text{SELECT}(i)$. If these 1s are evenly spread, substituting $\text{SELECT}(\sigma_l \cdot \ell) = 63488 \cdot s + a$, then $\text{SELECT}(i)$ is located at position

$$p = 63488 \cdot s + a + \frac{b - a}{\sigma_l} \cdot (i - \sigma_l \cdot \ell). \quad (1)$$
Thus, for any $i$, we define $p$ using Equation 1; we search for $\text{SELECT}(i)$ using a linear scan beginning at $p$. Specifically, since $\text{SELECT}(i)$ is contained in the $i$th superblock, we begin at basic block $B = \lfloor p/496 \rfloor$. We proceed with a linear scan. First, we ensure that the number of 1 bits before basic block $B$ is at most $i$. The number of 1 bits before block $B$ can be obtained by summing the local rank of $B$ with the $\lfloor B/128 \rfloor$th entry in the rank array. While this sum is at most $i$, we decrement $B$ and check again. Then, we use the same strategy to ensure that the number of 1 bits before block $B + 1$ is at least $i$ (while it is not, we increment $B$). After both loops complete, we know that $\text{SELECT}(i)$ is in block $B$.

Now, let us revisit the assumptions we made when defining predictions: what happens when $i$ is not well defined? Even on very well-behaved data, our low-level select array entries will cross superblock boundaries relatively frequently (see Figure 1b); we must ensure that in this case our queries are correct, and that our predictions are as accurate as possible.

At first glance, correctness appears to be guaranteed since we do a linear scan for the correct basic block – however, our algorithm as presented is not correct if $b < a$. In particular, if $b < a$ and $i$ is in the first or last superblock, we may have $p < 0$ or $p > n - 1$. On top of the correctness issue, $b < a$ also causes particularly bad predictions. We found that this case is common enough to noticeably slow down our queries.

Thus, if $b < a$, we make the following adjustment to ensure correctness and improve the solution quality. In short, we can test which sampled 1 is in the same superblock as $i$, and adjust $a$ or $b$ accordingly. Specifically, we test if entry $s$ in the rank array is at least $\sigma_i \cdot \ell$. If so, $\text{SELECT}(\sigma_i \cdot \ell)$ is in a superblock before $i$, so $a$ is not in the same superblock as $i$; we calculate $a' = a - 63488$. Otherwise we set $b' = b + 63488$. In each case we calculate $p'$ using Equation 1 with $a'$ or $b'$, and begin the search at $B = \lfloor p'/496 \rfloor$. For example, in Figure 1b, $a'$ would be a negative number representing the slots between $a$ and the beginning of superblock $s$; thus, $p'$ is in the range shown in the figure.

This fix improves results in a good prediction if the sampled bits are in successive blocks. Furthermore, a case-by-case analysis shows that $p$ is now always between 0 and $n - 1$. Of course, this simple fix does not always result in a good prediction – if $a$ and $b$ are separated by multiple blocks, or if $i$ is not well-specified but $a < b$, then $p$ is unlikely to be accurate – but these cases are rare enough on real-world data as to not significantly impact performance.

After the above, we have found the basic block $B$ containing $\text{SELECT}(i)$. We perform a select within $B$ by first masking out the first 16 bits, and then using fast select (Section 1.1.1).

**Space.** The rank array requires $64 \cdot n/63488$ bits. The modified bit vector requires $512 \cdot n/496$ bits. Summing, we obtain 3.33% extra space. The high-level select array requires at most $64 \cdot n/63488$ bits; the low-level select array requires at most $16 \cdot n/4096/99$ bits. Thus, select requires .495% space. Summing, SPIDER requires a 3.83% space overhead.
2.3 Non-Interleaved SPIDER

For some use cases, it may not be possible to modify $V$. With these cases in mind, we define a version of spider that does not interleave metadata: it leaves the original bit vector untouched, and uses 3.62% space to help answer rank and select queries. We call this data structure Non-Interleaved SPIDER, or ni-spider.

Non-Interleaved SPIDER has further advantages, even for use cases where the bit vector may be modified. It is simpler: ni-spider uses a simpler one-level select method that is easier to implement, and it avoids several corner cases. Furthermore, we discuss in Section 2.4 that the non-interleaved rank metadata improves select performance on some datasets.

Non-Interleaved Rank Data Structure. The rank data structure and queries for non-interleaved SPIDER is essentially the same as pasta-wide as given by by Kurpicz [11] (however, the two data structures differ for select queries). Nonetheless, we describe the rank data structure for completeness, as well as for reference when describing select queries.

We partition the array into $n/65536$ contiguous superblocks of 65536 bits. The rank array consists of $n/65536$ 64-bit entries, where the $i$th entry stores $\text{Rank}(65536i−1)$.

Then, we store a second level rank array. We partition the bit vector into $n/512$ basic blocks of 512 bits. We define the local rank of block $i$ to be $\text{Rank}(512i−1)−\text{Rank}(65536\lfloor 512i/65536 \rfloor−1)$. We store this value in slot $i$ of the second level rank array. We retain $V$ as well to help with rank and select queries.

Rank Queries. To find $\text{Rank}(i)$, we begin by using the rank array to find $r_1 = \text{Rank}(\lfloor i/65536 \rfloor−1)$ (this can be done by looking up the $\lfloor i/65536 \rfloor$th position in the rank array). Then, we use the second level rank array to find $r_2 = \text{Rank}(\lfloor i/512 \rfloor−1)$. Finally, we use popcount to count the number of 1s before position $i$ within basic block $b_2$ of $V$ (see Section 1.1.1).

Non-interleaved Select Data Structure For non-interleaved SPIDER we use a single-level select array. We discuss why a single level is sufficient in Section 2.4.

We calculate a single sampling threshold $\sigma = 2\lceil \log_2(16384 \cdot n_1/n) \rceil$. The select array has at most $n/16384$ entries, each of 64 bits. Entry $i$ of the select array stores $\text{Select}(\sigma \cdot i)$.

Non-interleaved Select Queries To answer $\text{Select}(i)$, let $a$ be the value stored in the $\lfloor i/\sigma \rfloor$th entry of the select array, and let $b$ be the value stored in the $(\lfloor i/\sigma \rfloor+1)$th entry. Thus we predict the value of $\text{Select}(i)$ to be $p = a + \frac{k-a}{\sigma}(i−\sigma \lfloor i/\sigma \rfloor)$. Since we store a single select array, there is no issue when $a$ and $b$ span a superblock boundary.

We begin at basic block $B = \lceil p/512 \rceil$. We use the rank array and the second-level rank array to scan for the correct basic block. The number of 1s before $B$ can be obtained by adding the $[B/128]$th entry in the rank array to the $B$th entry in the second-level rank array; while this number is greater than $i$ we decrement $B$. We then increment $B$ while the $(B/128)−1)$st rank array entry plus the $(B+1)$st second-level rank array entry are smaller than $i$. Once $B$ is found, we select in a block using fast select (see Section 1.1.1).

Space Usage. Summing $64n/65536 + 16n/512 + 64n/16384$ we obtain 3.62% extra space.

5 The constant 16384 was chosen to remain under 4% space usage.
2.4 Comparing SPIDER and Non-Interleaved SPIDER Select Queries

Here we discuss some performance differences between SPIDER and Non-Interleaved SPIDER.

For rank queries, the interleaved strategy has strictly better cache performance. SPIDER incurs $\leq 1$ cache miss when accessing the rank array, and a cache miss when accessing the modified bit vector. Non-Interleaved SPIDER incurs $\leq 1$ cache miss for the rank array, a cache miss to access the second-level rank array, and another to access the bit vector.

Interleaving the ranks has an immediate impact on select queries. Non-Interleaved SPIDER does a linear search for the block containing $\text{SELECT}(i)$ in the second-level rank array, whereas SPIDER searches the modified bit vector.

This leads to an interesting tradeoff in cache performance. SPIDER incurs a cache miss each time it considers a new basic block. In contrast, Non-Interleaved SPIDER almost never incurs more than one cache miss in the second-level rank array, since 32 consecutive basic blocks fit in a cache line. However, Non-Interleaved SPIDER incurs an extra cache miss to access the bit vector itself. Thus, assuming that all array accesses result in a cache miss, if $p$ is $g$ basic blocks away from $\text{SELECT}(i)$, SPIDER incurs $1 + g$ cache misses. Non-Interleaved SPIDER incurs on average $2 + g/16$ cache misses.

Thus, the cache advantages of SPIDER’s rank queries also apply to select queries if the predictions give exactly the correct basic block. However, inaccurate predictions immediately lead to cache misses for SPIDER, whereas Non-Interleaved SPIDER’s cache performance is minimally affected by prediction quality. The break-even point is when each prediction is 1 basic block away from the correct basic block on average.

This tradeoff motivates our two-level select array for SPIDER. The two-level design leads to more computation and a more complicated data structure. In exchange, the two-level array samples far more 1 positions in the same space, increasing the prediction quality. This is particularly important on sparse data (see Figure 5).

In addition to the difference in cost for bad predictions, SPIDER has a secondary advantage from avoid a second-level rank array: the extra array uses up the cache, potentially causing other metadata to be evicted much earlier. This may cause Non-Interleaved SPIDER to incur more cache misses than SPIDER when $n$ is small.

2.5 SPIDER vs Non-Interleaved SPIDER Predictions

We ran experiments to compare SPIDER and Non-Interleaved SPIDER in more detail. In Table 3, we give results for the accuracy of the prediction of each method: the table gives the average number of incorrect basic blocks looked at by SPIDER and Non-Interleaved SPIDER on three datasets. We see that the predictions are generally very accurate; even on the sparse Protein dataset, the predictions are off by approximately one basic block. We also see that, as one might expect, real data is more difficult to predict than random synthetic data. Nonetheless, the results in Figure 3 show that even with this modest decrease in prediction quality, the algorithms retain good results.

3 Experiments

We ran all experiments on a x86-64 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz with L1d, L1i, L2, and L3 cache sizes 192 KiB, 128 KiB, 5 MiB and 12 MiB respectively. The machine had 32GB RAM and was running Ubuntu 20.04 LTS. The code was written in C and compiled using gcc version 9.4.0 with compiler flags: -O9 -march=native -mpopcnt -mlzcnt.
Datasets. Our experiments use both synthetic and real data sets. For synthetic data, we provide three different densities: 10%, 50%, and 90%. The data is generated by setting each bit to 1 independently with probabilities 0.1, 0.5, or 0.9 respectively. We refer to these datasets as 10% Random, 50% Random, and 90% Random.

We give two bit vectors based on protein data, which we call Protein and Protein-Even. These bit vectors are based on the Uniref90 protein data set [26]. To generate Protein, we set each occurrence of a leucine amino acid (character L) to a 1 bit and all others to zero. Protein is sparse: 9.72% of the bits are 1s, and the rest are 0s. We also give experiments on a second dataset, which we call Protein-Even; this dataset was generated by setting characters A through L to 0 and all others to 1; this gives a density of 44.5% 1s.

The second dataset consists of a dump of all Wikipedia articles. We mapped all characters a through n and A through N to 1 and all others to 0; this resulted in a dataset with 47% 1s. The Wikipedia data has less than 32 billion characters, so results only go up to 16 billion.

We note that wavelet trees in particular motivate our Wikipedia and Protein-Even datasets: the top level in a wavelet tree is a bit vector that maps (roughly) half the original characters to 1 and the other half to 0 [18]. Thus, our Wikipedia and Protein-Even bit vectors are exactly the top level of a wavelet tree on the original wikipedia and Uniref 90 datasets.

Rank Query Performance. We compare spider and ni-spider to four other datasets. First, we compare to pasta-flat: this is the most performant data structure that can handle both rank and select queries with under 5% space. We also compare to pasta-wide, sds1-v,
and vigna-rank, all of which are tuned specifically for rank queries and cannot handle select queries efficiently (see Section 1.3). We note that ni-spider and pasta-wide are effectively the same method for rank (though not for select) queries, as are sds1-v and vigna-rank. Unsurprisingly, their performance values are very similar; any discrepancies are likely due to small implementation differences. Nonetheless, we include all results for completeness.

We give our rank results in Figure 2; results for the remaining datasets are essentially identical and are omitted. Overall, SPIDER gives the best rank results for large $n$, even compared to data structures that have worse space efficiency and are tailored specifically to rank queries. For smaller $n$, SPIDER is still competitive, though is not the best, likely due to the extra computation required to extract the interleaved metadata.
Select Query Performance. We compare spider and ni-spider to pasta-flat and pasta-flat-fs, the most performant data structures under 10% space, and vigna-select-H and vigna-select, which use over 10% space and cannot answer rank queries.

We give our select results in Figure 3. SPIDER achieves better performance than pasta-flat, the state of the art data structure using less than 5% space. Even compared to the space-inefficient data structures, SPIDER performs quite well, achieving roughly similar performance to vigna-select on 50% random and 90% random data, and performing only slightly worse than vigna-select on all other datasets. SPIDER does noticeably worse than vigna-select-H on Wikipedia, Protein Even, and Random 50% data, but better on the sparse datasets – this is unsurprising since vigna-select-H is specifically tailored to perform well for data sets with \( n_1/n \approx .5 \).

SPIDER and Non-Interleaved SPIDER achieve similar select performance on all datasets.

Build Time. We ran experiments comparing build time. Preprocessing is very lightweight for SPIDER and Non-Interleaved SPIDER: the data structures only consist of a few arrays that can each be built with a linear scan through the data. We used popcount and fast select during our build to minimize the computation time.

We give the results for build on 50% Random data in Figure 4; SPIDER and Non-Interleaved SPIDER are significantly faster to build than any other data structure. We note that the pasta-flat and pasta-flat-fs lines overlap in this figure.

Comparison of SPIDER Variants. SPIDER and Non-Interleaved SPIDER differ in two ways: Non-Interleaved SPIDER does not interleave data, and uses a single-level select array. Here, we compare to the other two natural variants of SPIDER: spider-1L-select interlaces the ranks and keeps a 1-level select array, and ni-spider-2L-select does not interleave ranks, but has a 2-level select array. The results are given in Figure 5.

Overall, SPIDER and Non-Interleaved SPIDER have the best results. However, the other two variants often achieve similar or even non-negligibly better performance, particularly for small values of \( n \). One interesting point is that spider-1L-select has a single-level select array, but interleaves local ranks (and thus has a high cost for bad guesses) – for the Random 50% data the single-level select array is sufficient for high-quality predictions, whereas for the sparse, relatively-difficult-to-predict Protein bit vector it lags noticeably behind the other variants.
Figure 5 Average time for a select query in nanoseconds for the four different variants of SPIDER.

We note that the distinction between the 1-level and 2-level select arrays has no impact on rank performance. Therefore, the performance of spider-1L-select is identical to that of spider (and ni-spider-2L-select is identical to ni-spider) in Figure 2, so we do not give additional rank experiments here.

4 Conclusion

In this work we give SPIDER, a succinct data structure for rank and select. SPIDER interleaves metadata with the underlying bit vector to improve rank performance, and uses predictions and a two-level select array to improve select performance, while using only 3.83% additional space. For rank queries, SPIDER gives an up to 22% improvement in query speed over the state of the art, even comparing to data structures using much more space. For select queries, SPIDER is up to 41% faster than the state of the art when comparing only to data structures that use under 5% space. SPIDER is as little as 3.1% slower than the state of the art data structure (which uses 12.2% extra space and cannot answer rank queries). While these speedups represent our best results, SPIDER consistently outperforms data structures using similar space, and is competitive with state of the art data structures using more space. Thus, SPIDER makes significant progress on eliminating the tradeoff between space usage and query time for rank and select queries. We show that SPIDER is effective on both real and synthetic datasets.

Our second data structure, Non-Interleaved SPIDER, leaves the underlying bit vector unchanged. Non-Interleaved SPIDER roughly matches the performance of the best known rank data structure other than SPIDER, and matches SPIDER select performance.
References


Efficient Computation of Topological Integral Transforms

Vadim Lebovici 1✉✉✉
Mathematical Institute, University of Oxford, United Kingdom

Steve Oudot ✉✉
Inria Saclay, Palaiseau, France

Hugo Passe ✉
Ecole Normale Supérieure de Lyon, France

Abstract
Topological integral transforms have found many applications in shape analysis, from prediction of clinical outcomes in brain cancer to analysis of barley seeds. Using Euler characteristic as a measure, these objects record rich geometric information on weighted polytopal complexes. While some implementations exist, they only enable discretized representations of the transforms, and they do not handle weighted complexes (such as for instance images). Moreover, recent hybrid transforms lack an implementation.

In this paper, we introduce eucalc, a novel implementation of three topological integral transforms – the Euler characteristic transform, the Radon transform, and hybrid transforms – for weighted cubical complexes. Leveraging piecewise linear Morse theory and Euler calculus, the algorithms significantly reduce computational complexity by focusing on critical points. Our software provides exact representations of transforms, handles both binary and grayscale images, and supports multicores processing. It is publicly available as a C++ library with a Python wrapper. We present mathematical foundations, implementation details, and experimental evaluations, demonstrating eucalc’s efficiency.

2012 ACM Subject Classification Mathematics of computing → Algebraic topology

Keywords and phrases Topological data analysis, Euler calculus, Topological integral transform, Euler characteristic transform, Hybrid transforms

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.22

Supplementary Material Software (Source Code): https://github.com/HugoPasse/Eucalc
archived at swh:1:dir:cee8afe463352094a26fc586f6cc1c92913de409

Funding Vadim Lebovici: Partly funded by EPSRC EP/R018472/1.

1 Introduction

1.1 Motivations
Since its introduction in the late 80s by Viro [17] and Schapira [13], Euler calculus – the integral calculus with respect to the Euler characteristic – has been of increasing interest in topological data analysis. Using this integral calculus, one can define integral transforms conveying significant topological information. The most famous one, the Euler characteristic transform (ECT), has found many applications in shape analysis, for instance in the prediction of clinical outcomes in brain cancer [5], in the analysis of barley seeds [2] or in the recovery of morphological variations across genera of primates [15]. More formally, given a sufficiently

1 For the purpose of Open Access, the author has applied a CC BY public copyright licence to any Author Accepted Manuscript (AAM) version arising from this submission.
tame compact shape \( K \) in Euclidean space (e.g., an embedded polytopal complex), the ECT records the Euler characteristic of the intersection \( \xi^{-1}(\langle -\infty, t \rangle \cap K) \) for all linear forms \( \xi : \mathbb{R}^n \rightarrow \mathbb{R} \) and all \( t \in \mathbb{R} \). What is remarkable is that this transform entirely characterizes the shape \( K \), that is, it is injective: if two shapes have same ECT, then they are equal [7, 8, 14, 16]. The ECT is an instance of a more general transform called the Radon transform [14] and the injectivity of the ECT is a consequence of Theorem 3.1 in loc. cit.. Another natural specialization of the Radon transform, simply called Radon transform in this paper, records the Euler characteristic of \( \xi^{-1}(t) \cap K \) for all linear forms \( \xi : \mathbb{R}^n \rightarrow \mathbb{R} \) and all \( t \in \mathbb{R} \).

The most problematic aspect of Euler calculus is its instability under numerical approximations [6]. To remedy it, integral transforms mixing Euler calculus and Lebesgue integration were introduced in [9], then generalized under the name hybrid transforms (HT) in [11]. These transforms integrate the Radon transform against a chosen kernel to smooth out and compress the information. The use of a kernel provides a wide variety of potential summaries that can be used to emphasize different aspects of data [10]. All these transforms are defined more generally for any constructible function, but in this article we restrict ourselves to the case of weighted polytopal complexes, i.e., polytopal complexes for which each cell is assigned an integer.

1.2 Existing implementations and limitations

In this paper, we are interested in the computation of topological integral transforms. Several implementations already exist. The Python package demeter [1, 2] implements a discretized version of the ECT for axis-aligned cubical complexes. This package turns an \( n \)-dimensional binary image into an embedded cubical complex \( C \) in \( \mathbb{R}^n \) in a prescribed way: vertices are defined by non-zero pixels and higher dimensional cubes are defined by adjacency relations. The ECT of \( C \) is then output for any direction \( \xi : \mathbb{R}^n \rightarrow \mathbb{R} \) as a discretization of the Euler characteristic curve \( t \in \mathbb{R} \mapsto \chi(\xi^{-1}(\langle -\infty, t \rangle \cap C)) \) for some chosen discretization step \( T \). The computation of the curve is done using the classical formula for the Euler characteristic \( \chi(\xi^{-1}(\langle -\infty, t \rangle \cap C)) \) as a sum of terms \((-1)^{\dim(c)}\) over all cells \( c \) of \( C \) such that \( c \subseteq \xi^{-1}(\langle -\infty, t \rangle) \). A bucket sort of \( \xi \) values on the vertices optimizes the computation of the discretization. The Python package Sinatra-Pro [15] implements a similar algorithm for triangular meshes both for the ECT and its smoothed version (SECT) [5].

These implementations suffer from several essential limitations. First of all, they only work with polytopal complexes, not weighted polytopal complexes. This prevents, e.g., the processing of images, which are ubiquitous in applications. Furthermore, these implementations only output discretized Euler characteristic curves. This is important because, while \( 2^n \) well-chosen Euler characteristic curves are sufficient to fully reconstruct a cubical complex embedded in \( \mathbb{R}^n \), each curve must be computed exactly for the reconstruction to be correct. Reconstructing the complex from discretized curves will likely lead to substantial errors in the reconstruction, due to the instability of Euler calculus with respect to numerical approximations. On the contrary, an exact computation of integral transforms would pave the way for the implementation of their left-inverses. A software computing an exact version of the ECT on weighted cubical complexes together with an algorithm for its left-inverse does exist, but only for two- and three-dimensional complexes [4]. Besides, this software (as all aforementioned ones) does not allow for the computation of the Radon and hybrid transforms. There is only one recently released software [10] that computes hybrid transforms for some specific complexes arising in the context of multi-parameter persistence. However, no algorithm exists for general polytopal complexes, nor even for cubical ones. To the best of our knowledge, no implementation of the Radon transform exists.
1.3 Contributions

We introduce fast implementations of the ECT, Radon transform, and hybrid transforms for weighted (axis-aligned) cubical complexes built from grayscale images. After a pre-processing step, our implementation can evaluate the transform exactly at any vector $\xi$ in time at most linear (and often largely sublinear in practice) in the number of vertices in the complex. This is thanks to two major optimizations.

Our first and main optimization comes from the use of piecewise linear Morse theory [3], which guarantees that, under some assumptions that are automatically satisfied by linear forms $\xi : \mathbb{R}^n \to \mathbb{R}$ on polytopal complexes $K$, the topology of $\xi^{-1}(-\infty,t]\cap K$ changes only when $t \in \mathbb{R}$ belongs to a finite set of so-called critical values. These critical values only appear when $\xi^{-1}(t)$ contains a vertex of the complex called a critical point. The set of critical values of $\xi$ together with the associated changes in the Euler characteristic of sublevel-sets yield a finite and exact representation of the associated Euler characteristic curve – see for instance [7] for a precise statement in the case of unweighted cubical complexes. This reduces the computation from a sum over all cells to a sum over only a few critical points. As an illustration, we show that there are between 34 and 21 times fewer critical points than vertices on the images of the fashion_MNIST data set.

Our second optimization comes from the observation that two linear forms $\xi$ and $\xi'$ whose canonical coordinates have same signs share the same critical points on axis-aligned cubical complexes. Precomputing these points for $2^n$ well-chosen directions enables fast evaluation of topological transforms in many directions at a very low cost. Moreover, the formalism of Euler calculus suggests that the results of piecewise linear Morse theory also hold for weighted cubical complexes. We exploit this remark in our implementation. Consequently, computing topological transforms of grayscale images is done in less than twice the time for binary ones.

Our software, called eucalc [12], is available as a C++ library together with a Python wrapper to make it easily compatible with scikit-learn like interfaces. It provides a data structure for weighted embedded cubical complexes allowing for fast preprocessing of critical points. Our data structure builds on the implementation of cubical complexes from the GUDHI library to benefit from their flexibility of instantiation. Moreover, our structure represents exact versions of the ECT and the Radon transform, together with evaluation and discretization routines. Our implementation also includes a parallelized version of the preprocessing step and the computation of the hybrid transform.

2 Preliminaries

Let $n \geq 2$ be an integer. The space of linear forms on $\mathbb{R}^n$ will be canonically identified with $\mathbb{R}^n$. For clarity, we refer to linear forms with greek letters $\xi \in \mathbb{R}^n$, and we denote the canonical dot product of $\xi$ and $x \in \mathbb{R}^n$ by $\langle \xi ; x \rangle$. For any function $f : \mathbb{R}^n \to \mathbb{R}$ and $t \in \mathbb{R}$, we denote by $\{ f = t \}$ and $\{ f \leq t \}$ the level set $\{ x \in \mathbb{R}^n : f(x) = t \}$ and the sublevel set $\{ x \in \mathbb{R}^n : f(x) \leq t \}$ respectively.

2.1 Weighted polytopal complexes

While our algorithms deal with cubical complexes, we introduce our main objects in the more general setting of polytopal complexes for the sake of clarity. We refer to Ziegler [18] for more details on polytopal complexes. We call polytope a bounded intersection of a finite number of closed half-spaces of $\mathbb{R}^n$. Equivalently, a polytope is the convex hull of a finite
number of points in $\mathbb{R}^n$, called its vertices [18, Thm. 1.1]. We denote by $\text{Vert}(P)$ the (finite) set of all vertices of a polytope $P$, and we call it its vertex set. A face of $P$ is any set of the form $F = P \cap \{ x \leq t \}$ where $x \in \mathbb{R}^n$ and $t \in \mathbb{R}$ are such that $P \subseteq \{ x \leq t \}$. The dimension of a polytope is the dimension of the smallest affine subspace of $\mathbb{R}^n$ containing it. A polytopal complex $\mathcal{P}$ is a finite collection of polytopes in $\mathbb{R}^n$ (called cells) such that (i) the empty cell is in $\mathcal{P}$, (ii) if $P$ is in $\mathcal{P}$, then all the faces of $P$ are also in $\mathcal{P}$, and (iii) if $P$ and $Q$ are in $\mathcal{P}$, then their intersection is a face of both $P$ and $Q$. One can always extend a collection of polytopes satisfying (i) and (iii) to a polytopal complex by adding all faces of the polytopes in the collection. This polytopal complex is the smallest one containing the polytopes of the given collection.

In this paper, we are mainly interested in cubical complexes. A cube in $\mathbb{R}^n$ is a subset $P = [a_1, b_1] \times \ldots \times [a_n, b_n] \subseteq \mathbb{R}^n$ such that $b_i - a_i = b_j - a_j$ if $a_i \neq b_i$ and $a_j \neq b_j$. A cubical complex $\mathcal{P}$ is a polytopal complex whose polytopes are cubes in $\mathbb{R}^n$.

Let $\mathcal{P}$ be a polytopal complex in $\mathbb{R}^n$ and $\xi \in \mathbb{R}^n$. The star of a polytope $P \in \mathcal{P}$ is the collection $\text{St}(P)$ of polytopes $Q \in \mathcal{P}$ such that $Q \supseteq P$. For any $x \in \text{Vert}(\mathcal{P})$, the lower star of $x$ with respect to $\xi$ is the collection $\text{LwSt}(\xi, x)$ of polytopes $Q \in \text{St}(x)$ such that $\max Q \xi = \xi(x)$. Similarly, we define the upper star $\text{UpSt}(\xi, x)$. These collections of polytopes are not complexes as they are not closed under taking faces.

Let $\mathcal{X}$ be a finite collection of polytopes in $\mathbb{R}^n$ and let $\varphi : \mathcal{X} \rightarrow \mathbb{Z}$. If $\mathcal{X}$ is a polytopal complex, then we call $\varphi$ a weighted polytopal complex. The Euler characteristic of $\varphi$ is defined as $\chi(\varphi) = \sum_{P \in \mathcal{X}} (-1)^{\dim(P)} \varphi(P)$. This definition generalizes the usual notion of Euler characteristic of a polytopal complex [18, Corollary 8.17] and is a particular case of Euler integration [17, 13].

### 2.2 Integral transforms

Let $\xi \in \mathbb{R}^n$ and $t \in \mathbb{R}$. Consider the collections of polytopes $\{ P \cap \{ x \leq t \} : P \in \mathcal{P} \}$ and $\{ P \cap \{ x = t \} : P \in \mathcal{P} \}$ and denote respectively by $\mathcal{P}_{\{ x \leq t \}}$ and $\mathcal{P}_{\{ x = t \}}$ the smallest polytopal complexes containing these collections. Any function $\varphi : \mathcal{P} \rightarrow \mathbb{Z}$ induces a function $\varphi_{\{ x \leq t \}} : \mathcal{P}_{\{ x \leq t \}} \rightarrow \mathbb{Z}$ defined for any $Q \in \mathcal{P}_{\{ x \leq t \}}$ by $\varphi_{\{ x \leq t \}}(Q) = \varphi(P)$ where $P$ is the smallest polytope of $\mathcal{P}$ containing $Q$. In that case, one has $\dim(Q) = \dim(P) - 1$. Similarly, the function $\varphi$ induces a function $\varphi_{\{ x = t \}} : \mathcal{P}_{\{ x = t \}} \rightarrow \mathbb{Z}$.

Let $\varphi : \mathcal{P} \rightarrow \mathbb{Z}$. The first topological integral transform we consider in this paper is the Radon transform [14], namely, the function $\mathcal{R}_{\varphi} : \mathbb{R}^n \times \mathbb{R} \rightarrow \mathbb{Z}$ defined for any $(\xi, t) \in \mathbb{R}^n \times \mathbb{R}$ by:

$$\mathcal{R}_{\varphi}(\xi, t) = \chi \left( \varphi_{\{ x = t \}} \right).$$

The second integral transform, the Euler characteristic transform [16], is the function $\text{ECT}_{\varphi} : \mathbb{R}^n \times \mathbb{R} \rightarrow \mathbb{Z}$ defined for any $(\xi, t) \in \mathbb{R}^n \times \mathbb{R}$ by:

$$\text{ECT}_{\varphi}(\xi, t) = \chi \left( \varphi_{\{ x \leq t \}} \right).$$

Following [11], we consider specific smoothings of topological transforms. Let $\kappa : \mathbb{R} \rightarrow \mathbb{C}$ be locally integrable. The hybrid transform of $\varphi$ with kernel $\kappa$ is the function $\text{HT}_{\varphi} : \mathbb{R}^n \rightarrow \mathbb{C}$ defined for any $\xi \in \mathbb{R}^n$ by:

$$\text{HT}_{\varphi}(\xi) = \int_{\mathbb{R}} \kappa(t) \mathcal{R}_{\varphi}(\xi, t) dt.$$
2.3 Naive algorithm

The following lemma is an easy consequence of the definitions.

▶ Lemma 1. Let $\bar{\kappa} : \mathbb{R} \rightarrow \mathbb{C}$ be a primitive of $\kappa$.

$$R_\varphi(\xi, \cdot) = \sum_{P \in \mathcal{P}} \varphi(P) \mathbf{1}_{[\min_P \xi, \max_P \xi]},$$

$$ECT_\varphi(\xi, \cdot) = \sum_{P \in \mathcal{P}} \varphi(P) \mathbf{1}_{[\min_P \xi, +\infty)},$$

As a consequence,

$$HT_\varphi(\xi) = \sum_{P \in \mathcal{P}} \varphi(P) \left( \bar{\kappa} \left( \max_P \xi \right) - \bar{\kappa} \left( \min_P \xi \right) \right).$$

This lemma straightforwardly suggests naive algorithms to compute topological integral transforms over any polytopal complex. Doing so, the time complexity of computing any one of the above integral transforms over $\mathcal{P}$ is $O(N \cdot \#P)$ where $N$ is the maximum number of vertices of cells in $\mathcal{P}$. These naive algorithms will serve as a baseline for our future optimizations. Notice that $\#P$ is exponential in the dimension.

3 Euler critical points

In this section, we slightly generalize the notion of Euler critical value of polytopal complexes [7, Def. 6.2] to weighted polytopal complexes. All mathematical results are folklore. We state them and provide proofs in Appendix B for completeness.

Let $\mathcal{P}$ be a polytopal complex, $\xi \in \mathbb{R}^n$ and $\varphi : \mathcal{P} \rightarrow \mathbb{Z}$. Further assume that $\xi$ is generic on $\mathcal{P}$, i.e., no polytope of $\mathcal{P}$ of positive dimension is contained in a level set of $\xi$. We define the ordinary and classical critical values of $\xi$ at a vertex $x$ of $\mathcal{P}$ respectively by:

$$\Delta_\varphi^{\text{ord}}(\xi, x) = \chi(\varphi\mid_{\text{UpSt}(\xi, x)}) - \chi(\varphi\mid_{\text{LwSt}(\xi, x)}),$$

$$\Delta_\varphi(\xi, x) = \chi(\varphi\mid_{\text{LwSt}(\xi, x)}).$$

A vertex $x$ of $\mathcal{P}$ is an Euler $\varphi$-critical point of $\xi$ if one of $\Delta_\varphi^{\text{ord}}(\xi, x)$ or $\Delta_\varphi^{\text{ord}}(\xi, x)$ is non-zero. Such a point is called ordinary when $\Delta_\varphi^{\text{ord}}(\xi, x) \neq 0$ and classical when $\Delta_\varphi(\xi, x) \neq 0$. Euler $\varphi$-critical points are sometimes called critical points when $\varphi$ and $\xi$ are clear from the context. Note that a critical point can be both classical and ordinary. The set of classical critical points is denoted by $\text{Crit}_\varphi^{\text{ord}}(\xi)$, the set of ordinary critical points by $\text{Crit}_\varphi^{\text{ord}}(\xi)$ and the set of all critical points by $\text{Crit}_\varphi(\xi) = \text{Crit}_\varphi^{\text{ord}}(\xi) \cup \text{Crit}_\varphi^{\text{ord}}(\xi)$. Critical points and values are illustrated in Appendix A. The following lemma expresses integral transforms as sums over critical points. It is key to our algorithms.

▶ Lemma 2. Let $\varphi : \mathcal{P} \rightarrow \mathbb{Z}$ and let $\xi \in \mathbb{R}^n$ be generic on $\mathcal{P}$. Then,

$$R_\varphi(\xi, \cdot) = \sum_{x \in \text{Crit}_\varphi(\xi)} \Delta_\varphi(\xi, x) \mathbf{1}_{[(\xi(x), +\infty)} + \sum_{x \in \text{Crit}_\varphi^{\text{ord}}(\xi)} \Delta_\varphi^{\text{ord}}(\xi, x) \mathbf{1}_{[(\xi(x), +\infty)},$$

$$ECT_\varphi(\xi, \cdot) = \sum_{x \in \text{Crit}_\varphi(\xi)} \Delta_\varphi(\xi, x) \mathbf{1}_{[(\xi(x), +\infty).}$$

Furthermore, if $\kappa : \mathbb{R} \rightarrow \mathbb{C}$ is locally integrable and $\bar{\kappa}$ is a primitive of $\kappa$ on $\mathbb{R}$, one has:

$$HT_\varphi(\xi) = -\sum_{x \in \text{Crit}_\varphi^{\text{ord}}(\xi)} \Delta_\varphi^{\text{ord}}(\xi, x) \cdot \bar{\kappa}(\xi(x)) \cdot \bar{\kappa}(\xi(x)).$$
Let $C$ be a cubical complex. The result below can be adapted to arbitrary polytopal complexes but we do not make use of such a general statement in our algorithms. Since our definition of cubes only allow for axis-aligned cubes, any $\xi \in \mathbb{R}^n$ with non-zero coordinates is generic on $C$. For any $\xi = (\xi_1, \ldots, \xi_n) \in \mathbb{R}^n$ with non-zero coordinates we define its sign vector $\varepsilon^\xi \in \{\pm 1\}^n$ by $\varepsilon^\xi_i = \text{sgn}(\xi_i)$. The following lemma ensures that only $2^n$ sets of critical points and values are necessary to compute integral transforms in practice.

**Lemma 3.** Let $\xi, \xi' \in (\mathbb{R} \setminus \{0\})^n$. If $\varepsilon^\xi = \varepsilon^\xi'$, then for any $\varphi : C \to \mathbb{R}$ and any $x \in \text{Vert}(C)$ the sets and numbers $\text{LwSt}(\xi, x)$, $\text{UpSt}(\xi, x)$, $\Delta^-_{\varphi}(\xi, x)$, $\Delta^{\text{ord}}_{\varphi}(\xi, x)$, $\text{Crit}^-_{\varphi}(\xi)$ and $\text{Crit}^{\text{ord}}_{\varphi}(\xi)$ are respectively equal to the ones associated to $\xi'$.

### 4 Implementation and asymptotic analysis

The mathematical results of the previous section suggest algorithms allowing for efficient computation of the three integral transforms under consideration in this paper. In this section, we describe our implementation and provide worst-case time complexity bounds.

#### 4.1 Implementation of cubical complexes

Our implementation of cubical complexes consists in a C++ class `EmbeddedComplex` that inherits from the cubical complex class `Bitmap_cubical_complex` from the GUDHI library (release 3.8.0) and additionally stores the embedding coordinates of the vertices in $\mathbb{R}^n$. This inheritance allows us to take any $n$-dimensional array/image as input and use GUDHI’s initialization to turn it into a cubical complex. The GUDHI library stores the weighted cubical complex as a vector of bits representing the value assigned to each cell, with a specific convention for the indexing of cells. By default, one top-dimensional cell is associated to each pixel and the values of lower-dimensional cells are equal to the minimum of the values of their cofaces. By convention, the embedding coordinates of the vertices in $\mathbb{R}^n$ are evenly spaced and normalized in $[-0.5, 0.5]^n$. Storing the values of the complex in GUDHI’s initialization is linear in the number of cells in the complex. We do not consider it as part of our algorithm. However, we take it into account in the running times in our experiments of Section 5.

#### 4.2 Preprocessing of critical points and values

After the initialization, the first step to compute integral transforms using Lemma 2 is to compute the critical values $\Delta^-_{\varphi}(\varepsilon, x)$ and $\Delta^{\text{ord}}_{\varphi}(\varepsilon, x)$ for all $x \in \text{Vert}(C)$ and all sign vectors $\varepsilon \in \{\pm 1\}^n$. While doing so, we store on the fly each one of the families $\Delta^-_{\varphi}(\varepsilon) := \{\Delta^-_{\varphi}(\varepsilon, x)\}_{x \in \text{Crit}_{\varphi}^-}$ and $\Delta^{\text{ord}}_{\varphi}(\varepsilon) := \{\Delta^{\text{ord}}_{\varphi}(\varepsilon, x)\}_{x \in \text{Crit}_{\varphi}^{\text{ord}}}$ as a pair of two arrays, one for the critical points and one for the associated critical values, which induces no overhead in terms of asymptotic running time. Using GUDHI’s implementation of weighted cubical complexes, accessing the dimension $\text{dim}(\varepsilon)$ and the value $\varphi(\varepsilon)$ of a cell $\varepsilon$ takes constant time. Moreover, cells are indexed by a single number in this implementation. As a consequence, iterating through the $O(2^n)$ adjacent cells of a vertex requires arithmetic operations to compute indices of adjacent cells, resulting in $O(n2^n)$ time. The computation of critical values at a vertex thus has the same complexity, and the overall time complexity of computing critical values for all vertices and all sign vectors is then $O(n4^n \cdot \text{#Vert}(C))$. 


4.3 Exact representation and evaluation

After the preprocessing step, the computation of the transforms using Lemma 2 goes as follows. Let \( \xi \in \mathbb{R}^n \), and denote \#Crit\(\overline{\varphi}(\varepsilon^t) \) and \#Crit\(\overline{\varphi}(\varepsilon^t) \) respectively by \( N^- \) and \( N^{ord} \).

First, we compute the sign vector \( \varepsilon^t \) in \( \mathcal{O}(n) \) time, which will always be negligible.

To compute the hybrid transform \( HT_{\varphi}(\xi) \), we sum, over all ordinary critical points \( x \in Crit_{\varphi}(\varepsilon^t) \), the evaluation – assumed to be done in constant time – of the kernel \( \bar{\kappa} \) on the dot products \( \langle \xi; x \rangle \). Hence a running time in \( \mathcal{O}(nN^{ord}) \).

To compute the Euler characteristic transform \( ECT_{\varphi}(\xi, \cdot) \), we first compute \( T = \{ \langle \xi; x \rangle \}_{x \in Crit_{\varphi}(\varepsilon^t)} \) in \( \mathcal{O}(nN^-) \) time. Then, we sort the set \( T \) and the sets \( \text{Crit}_{\varphi}(\varepsilon^t) \) and \( \Delta_{\varphi}(\varepsilon^t) \) by ascending values of \( \langle \xi; x \rangle \) in \( \mathcal{O}(N^- \log N^-) \) time. Finally, we compute a vector \( E \) that contains the cumulative sums of the sorted values \( \Delta_{\varphi}(\varepsilon^t) \) in \( \mathcal{O}(N^-) \) time. An exact representation of the function \( ECT_{\varphi}(\xi, \cdot) \) is then returned as the pair of vectors \( (T, E) \). The total running time is in \( \mathcal{O}(N^- (n + \log N^-)) \).

To compute the Radon transform, we do the same as for the ECT but replacing \( \text{Crit}_{\varphi}(\varepsilon^t) \) by \( \text{Crit}^{ord}_{\varphi}(\varepsilon^t) \) and \( \Delta_{\varphi}(\varepsilon^t) \) by \( \Delta^{ord}_{\varphi}(\varepsilon^t) \), to get an analogously constructed pair \( (T, E') \) in \( \mathcal{O}(N^{ord}(n + \log N^{ord})) \) time. Then, we sort the elements \( x \in Crit_{\varphi}(\varepsilon^t) \) by ascending values of \( \langle \xi; x \rangle \) in \( \mathcal{O}(N^- (n + \log N^-)) \) time. After this, we use binary search to find, for each \( x \in Crit_{\varphi}(\varepsilon^t) \), the critical point \( y_x \in Crit^{ord}_{\varphi}(\varepsilon^t) \) with highest value \( \langle \xi; y_x \rangle < \langle \xi; x \rangle \). This takes \( \mathcal{O}(N^- \log N^{ord}) \) time. We store \( E'[y_x] + \Delta^{ord}_{\varphi}(\varepsilon^t, x) \) for each \( x \in Crit_{\varphi}(\varepsilon^t) \) in an array \( E_c \). If there exist several points \( x' \in Crit_{\varphi}(\xi) \) with the same dot product \( \langle \xi; x' \rangle \), we only store the cumulative sum of the critical values \( \Delta^{ord}_{\varphi}(\varepsilon^t, x') \) in one entry of \( E_c \). An exact representation of the function \( R_{\varphi}(\xi, \cdot) \) is then returned as the tuple of the four sorted vectors \( T, E', T_c = \{ \langle \xi; x \rangle \}_{x \in Crit_{\varphi}(\varepsilon^t)} \) and \( E_c \), in \( \mathcal{O}(N^{ord}, n + (N^{ord} + N^-) \cdot \log N^{ord}) \) time.

Remark 4. To better understand these complexities, note that the number of vertices of \( \mathcal{C} \) is asymptotically equivalent to the number of pixels in the image. However, the total number of vertices appears only in the complexity of the preprocessing step. In many applications, the numbers \( N^{ord} \) and \( N^- \) are small compared to \#Vert(\mathcal{C}) so that the computation step is significantly optimized. This heuristic will be confirmed in Section 5.

To evaluate the Euler characteristic transform \( ECT_{\varphi}(\xi, \cdot) \) at a point \( t \in \mathbb{R} \) using its exact representation \( (T, E, T_c, E_c) \), we use a binary search to find the element \( x \in Crit^{ord}_{\varphi}(\varepsilon) \) with greatest \( \langle \xi; x \rangle \leq t \) and return \( E[x] \) in \( \mathcal{O}(\log N^{ord}) \). We also implemented a vectorization routine to get the values of the ECT at sorted and evenly spaced points inside a specified interval. This is done by iterating simultaneously over \( T \) and over the points of the interval, hence a running time in \( \mathcal{O}(N + N^{ord}) \).

The algorithm to evaluate the Radon transform using its exact representation \( (T, E', T_c, E_c) \) is identical, except when there exists \( x \in Crit^{ord}_{\varphi}(\varepsilon) \) such that \( t = \langle \xi; x \rangle \), in which case it returns \( E_c[x] \). Again, we use binary search to determine the existence of such a point \( x \) in \( \mathcal{O}(\log N^-) \) time.

5 Experimental evaluation

In this section, we illustrate our optimizations on several batches of experiments. In all experiments, integral transforms are computed in 100 directions drawn uniformly at random in the cube \( [0, 1]^n \). We compute hybrid transforms with \( \kappa(t) = \sin(t) \). To allow for a comparison with demeter, we instantiate our cubical complexes using the dual construction, that is, initializing the values of the vertices with the pixel values in the images, and setting the values of higher-dimensional cells as the minimum of the values of their vertices. Moreover,
we compare our exact version of the ECT to the discretized version of demeter with resolution $T=100$. This is justified by the fact that our vectorization step turning our exact version to a discretized one has negligible cost compared to the other steps, as shown in Figure 4.

Our code has been compiled with g++ 9.4.0 and run on a workstation with an Intel(R) Core(TM) i7-4770 CPU (8 cores, 3.4GHz) and 8 GB RAM, running GNU/Linux (Ubuntu 20.04.1). We measure time consumption using the Python module time. The input data and the benchmark Python scripts are available upon request.

5.1 Data set example

To begin with this experimental section, we compute integral transforms on a real-world data set: fashion_MNIST. This data set contains 60000 grayscale images of size $28 \times 28$ representing several types of clothes. We chose this data set because it contained global geometric information, namely the shape of clothes.

Results are shown in Table 1. In the first three lines, we timed our implementation of integral transforms on the fashion_MNIST data set with unchanged pixel values, potentially ranging from 0 to 255. In the next four lines, we timed our implementation as well as the ECT of demeter on a binarized version of the fashion_MNIST data set, where the 50% brightest pixels of each image are set to 1 and all other pixels to 0. This binarization was necessary since demeter does not cope with grayscale images.

What stands out from this experiment is that all transforms can be computed in a reasonable amount of time on this large data set. The preprocessing step seems to be even more efficient in the case of hybrid transforms, where the evaluation step in 100 directions is done in between a third and a sixth of the time of the preprocessing step.

More importantly, this experiment perfectly illustrates the efficiency of our implementation. Each $28 \times 28$ image of the fashion_MNIST data set induces a cubical complex with 784 vertices. On average, there are 172 classical critical points with a standard deviation of 50 and 237 ordinary critical points with a standard deviation of 65. Thus, around a quarter of the vertices are classical critical points, and the same is true for ordinary critical points. In contrast, the complexes of the binarized data set have 23 classical critical points on average with a standard deviation of 12 and 36 ordinary critical points with a standard deviation of 19. As a consequence, after the initialization and preprocessing steps, our optimization using critical points allows us to compute an exact version of the ECT on the binarized version more than 50 times faster than demeter computes its discretized version. Overall, our implementation is 18 times faster than demeter.

<table>
<thead>
<tr>
<th>Software</th>
<th>Init.</th>
<th>Preproc.</th>
<th>Comp.</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radon</td>
<td>13</td>
<td>143</td>
<td>290</td>
<td>447</td>
</tr>
<tr>
<td>HT</td>
<td>13</td>
<td>91</td>
<td>32</td>
<td>137</td>
</tr>
<tr>
<td>ECT</td>
<td>13</td>
<td>54</td>
<td>117</td>
<td>184</td>
</tr>
<tr>
<td>Radon (binary)</td>
<td>13</td>
<td>138</td>
<td>56</td>
<td>207</td>
</tr>
<tr>
<td>HT (binary)</td>
<td>13</td>
<td>88</td>
<td>14</td>
<td>115</td>
</tr>
<tr>
<td>ECT (binary)</td>
<td>13</td>
<td>52</td>
<td>32</td>
<td>97</td>
</tr>
<tr>
<td>demeter ECT</td>
<td>0</td>
<td>266</td>
<td>1544</td>
<td>1810</td>
</tr>
</tbody>
</table>

Table 1 Timing (s) of our implementation and of the demeter ECT on the fashion_MNIST data set.
Moreover, thanks to our generalization of Euler critical values to weighted polytopal complexes, it is still 10 times faster to compute our exact version of the ECT on the original grayscale data set than to compute it on the binarized data set using demeter.

5.2 Asymptotic analysis

In this section, we perform an experimental asymptotic analysis of our algorithms. This analysis confirms our theoretical results on time complexities of Section 4.

Size of images

We study the computation times on $m \times m$ images with respect to the parameter $m$. We use binary images with constant number of critical points made of a periodic pattern of black squares in a white background. The number of squares is constant when $m$ varies to ensure a constant number of critical points. Namely, they are 100 classical critical points and 200 ordinary critical points for all images.

As one could expect, the initialization (Figure 1a) seems linear in the number of cells in the complex, that is, quadratic in $m$. Similarly, the preprocessing step (Figure 1b) is linear in the number of vertices in the complex, that is, quadratic in $m$. The strength of our implementation is illustrated by Figure 1c. After the initialization and the preprocessing steps, computing the integral transforms is done in constant time when the number of critical points is fixed. The curves for our times in Figure 1c are constant and lie between 1 and 5 milliseconds.

Overall, our computation times are dominated by the preprocessing step. Remarkably, our implementation has improved computation times by a factor of 4 compared with demeter.

Number of critical points

We study the computation times with respect to the number of critical points in $1000 \times 1000$ images. We use the same periodic pattern of squares to control the number of critical points, this time with a varying number of squares.

The initialization and preprocessing steps are done in constant time for a constant size of image. More precisely the initialization takes around 1 second and the preprocessing of classical and ordinary critical points take approximately 3 and 6 seconds respectively. The “complexifying” step of demeter also takes around 6 seconds.
After the initialization and the preprocessing steps, computing the integral transforms is done in linear (HT) and linearithmic (ECT, Radon) times in the number of critical points for a fixed size of images. This is clearly illustrated in Figure 2a. Overall, computing an Euler characteristic curve for a binary image with four millions pixels takes between a 0.1 milliseconds to a few seconds depending on the number of critical points.

Interestingly, the computation time for the demeter package decreases with the number of critical points. This is due to our periodic patterns with varying size used to generate images with a varying number of critical points. The smaller the patterns are, the greater the number of critical points, but the lesser edges and two-dimensional cubes in the complex. As a consequence, when the number of critical points increases up to one quarter of the total number of vertices, demeter has a faster evaluation step than our evaluation of the ECT (Figure 2a). Time for the total pipeline are thus very similar between their and our implementation (Figure 2b).

However, this fact can be mitigated by two remarks. First of all, we compute an exact version of the Euler characteristic transforms and not an approximated version with a resolution of $T=100$. This approximated version is far from the exact one, as they are 25000 changes of values in the true Euler characteristic curve when they are 25000 classical critical points. Secondly, the number of critical points in real-world binary images containing global geometric features are likely to be way lower than a quarter of the vertices, as suggested by our study on a real-world data set in Section 5.1. While the number of critical points is likely to be higher for grayscale images, the demeter package can no longer be used in this case.

**Number of directions**

We study the influence of the number of requested directions on the total running time. All computations are done on a random binarized image of the fashion MNIST dataset. We run the computation of the ECT (including initialization and preprocessing) for 1 to 10000 directions sampled uniformly in $[0,1]^2$, and measure the corresponding times. The initial value of each curve corresponds to the initialization time. We give these plots on log-scale in Figure 3a, then on linear scale with the initial values forced to zero (which corresponds to ignoring the initialization time) in Figure 3b. By a linear regression, we check that the points we describe sample a linear function and we determine that its slope is $2.8 \cdot 10^{-6}$ seconds by direction for our method, and $2.3 \cdot 10^{-4}$ seconds by direction for demeter.
Vectorization

On Figure 4 we ran the vectorization routine on 1000x1000 image with $10^6$ critical points for a varying dimension of vectorization, that is, sampling the ECT on an increasing number of points $t \in \mathbb{R}$. The complexity of the vectorization routine is confirmed by Figure 4. More importantly, vectorization is negligible (a few milliseconds) compared to other steps of the computation of the transform (a few seconds each).

Average number of critical points

In this section, we expose an experimental study of the average number of critical points in random settings for two-dimensional grayscale images as the number of shades of gray increases. Pixel values are drawn using a uniform distribution in a first setting and using a Gaussian random field in a second setting.

In both cases, the number of both types of critical points in grayscale images is approximately half the number of vertices as soon as the number of pixel values is above 10. In the binary case, the number of critical points of each type is between 10% and 20% of the total number of vertices. These results corroborate the computation times observed on the data sets presented above: the number of critical points is in practice lower than the number of vertices, and much lower than the number of cells in the complex.
While this experiment shows that the number of critical point is of the same order as the number of vertices, it is expected not to be the case on digital images with global geometric features, as suggested in Section 5.1.

![Figure 5](image.png) Number of critical points in two random settings. Red and blue symbols represent respectively classical and ordinary critical points. Dots and crosses respectively represent these numbers for the uniform setting and for the Gaussian random field setting.

![Figure 6](image.png) Relative pre-processing time (s) given the number of cores used for several complex sizes.

### 5.3 Parallelization

We parallelized the routines `preproc_ordinary_pts` and `preproc_classical_pts` as they are the most time-consuming functions. To do so, we parallelized the inner loop on $\text{Vert}(C)$, each core handling the computations on a subset of the vertices. The readings are concurrent as cores only need to access fixed cell values. Each core returns an array of critical points and an array of critical values and the main core concatenates the results. Parallelizing the inner loop instead of the main one is efficient because the number of sign vectors in dimension $n$ is $2^n$, and $n = 2$ or $3$ in most practical cases. We also implemented a parallelization of the routine `compute_ht` that takes as input a vector of directions and returns a vector containing the values of the hybrid transforms in these directions. In this case each core handles the computations on a subset of the directions and returns the result to the main core that concatenates them. On Figure 6 we study the time of the preprocessing of critical points.
(both classical and ordinary) given the number of cores used, relatively to the single core setup. We do this for several complex sizes with cells values uniformly sampled in \{0, 1\}. We see that the parallelization induces a really small overhead, even for small complexes. Moreover, recall that even with only one core, our optimizations allow our software to be 100 times faster than the demeter package on images of the fashion_MNIST data set.

6 Conclusion

We presented a novel implementation of topological integral transforms on weighted cubical complexes, based on existing results from piecewise linear Morse theory slightly adapted to weighted polytopal complexes. These results allow for the fast computation of integral transforms on weighted cubical complexes. As a consequence, our implementation can efficiently compute exact versions of the ECT and Radon transform, and we showed experimentally that it is 18 times faster than the demeter ECT on a real-world data set, and 50 times faster after the preprocessing step. We also added a vectorization routine for data analysis purposes.

A natural extension of our implementation would be to arbitrary polytopal complexes. For any such complex, there is a cellular decomposition of (the dual of) \( \mathbb{R}^n \) such that Lemma 3 holds for any \( \xi, \xi' \) lying in the same cell [7]. This cellular decomposition is induced by the arrangement of hyperplanes that are orthogonal to the edges of the complex. The main challenge is that the size of this arrangement for a general polytopal complex with \( n \) vertices in \( \mathbb{R}^d \) may be as large as \( n^{2d} \), as opposed to only \( 2^d \) for cubical complexes.

References

2. Erik J Amézquita, Michelle Y Quigley, Tim Ophelders, Jacob B Landis, Daniel Koenig, Elizabeth Munch, and Daniel H Chitwood. Measuring hidden phenotype: Quantifying the shape of barley seeds using the euler characteristic transform. in silico Plants, 4(1), 2022.
In this section, we present an example of run of our algorithms on a small binary cubical complex. The complex is illustrated in Figure 7. Cells with non-zero values include 9 vertices, 9 edges, and 2 squares.

![Figure 7 Example of cubical complex. Colored cells have value 1 and light gray cells have value 0. The blue cells are vertices, the green ones are edges and the gray ones are faces. For greater clarity we display the keys of the vertices.](image)

### A.1 Preprocessing

In this section, we unfold the computation of the critical points. We only detail it for the ordinary ones, the computation for the classical ones being similar. We detail the operations for $\varepsilon = (1, -1)$ and $x = 32$. First we compute $e_-$, the euler characteristic of the lower star of $x$ in direction $\varepsilon$. The coordinates of $x$ are $c = (32 \div 7, 32 \mod 7) = (4, 4)$, so we explore all cells with coordinates $(4 - \delta_0 \varepsilon_0, 4 - \delta_1 \varepsilon_1)$ where $(\delta_0, \delta_1) \in \{0, 1\}^2$. These coordinates are $(4, 4), (3, 4), (4, 5)$ and $(3, 5)$. They correspond to the cells 32, 31, 39 and 38 respectively. Therefore, one has $e_- = -1 + 1 + 0 + 0 = 0$. In the same way we can compute $e_+$, the upper
We now unfold the evaluation of the Radon transform and ECT. In this example, we want

Apart from this, the computations are identical.

Thus \( e_+ \neq e_- \) and the vertex 32 is an ordinary critical point of \( \varepsilon = (1, -1) \) with critical value \(-1\). We list critical points and values for all \( \varepsilon \in \{-1, 1\}^2 \) in Table 2.

### A.2 Computation

Now, we unfold the computation step for Radon transform, ECT, and hybrid transform. Throughout the section, we set \( \xi = (2, 2) \) and denote its sign vector simply by \( \varepsilon = (1, 1) \). We first describe the computation of the ECT. We start by computing the dot product of \( \xi \) with all the critical points in \( C^\varepsilon_- = \{0, 28, 16, 30\} \). These critical points have coordinates \([-0.5, -0.5], [-0.5, 0.167], [-0.167, -0.167] \) and \([-0.167, 0.167] \) in \( \mathbb{R}^2 \). Thus the dot products are \([-2, 0.67, -0.67, 0]\). Then, we sort the critical points by increasing dot product, obtaining the list \( \text{sorted}(C^\varepsilon_-) = [0, 28, 16, 30] \). Finally, we sum the critical values \( \Delta^\varepsilon_- \) in the order given by the sorting of critical points. We begin with \( T = [-\infty] \) and \( E = [0] \). After visiting the first two vertices, 0 and 28, we obtain \( T = [-\infty, -2, -0.67] \) and \( E = [0, 1, 2] \). The next sorted vertex is 16, which has the same dot product as 28. In that case, we add the variation associated to 16 to the last element of \( E \). We obtain \( T = [-\infty, -2, -0.67] \) and \( E = [0, 1, 3] \). Finally, after adding vertex 30 we obtain \( T = [-\infty, -2, -0.67, 0] \) and \( E = [0, 1, 3, 2] \). The computation of the Radon transform follows the same process at first but with \( C^\varepsilon_0 \) instead of \( C^\varepsilon_- \). In that case, \( T = [-\infty, -0.67, 0.67, 1.33] \) and \( E' = [0.0, -2.0, -1.0, 0.0] \). For each \( x \in C^\varepsilon_0 \), we must then find the greatest \( i_0 \) such that \( T[i_0] \leq (\xi; x) \), and add \( (\xi; x) \) to \( T \) and \( E'[i_0] + \Delta^\varepsilon_0 \) to \( E \). We thus get \( T_{c} = [-2.0, -0.67, 0.0] \) and \( E_{c} = [1.0, 0.0, -1.0] \). Finally, computing the hybrid transform is done by summing \(-\kappa((\xi; x)) \cdot \Delta^\varepsilon_0 \) for \( x \in C^\varepsilon_0 \). In our case, with \( \kappa = \exp \), we obtain \( \text{HT}_\varepsilon(\xi) \simeq -4.715 \).

### A.3 Evaluation

We now unfold the evaluation of the Radon transform and ECT. In this example, we want to evaluate \( \text{ECT}_\varepsilon(\xi, t) \) for some \( t \in \mathbb{R} \). For \( \xi = (2, 2) \), we computed in the above section that \( T = [-\infty, -2, -0.67, 0] \) and \( E = [0, 1, 3, 2] \). By a binary search, we find the index \( i \) of the largest value in \( T \) that is smaller than \( t \). Then, we let \( \text{ECT}_\varepsilon(\xi, t) = E[i] \). For instance \( t = -0.3 \) gives \( i = 2 \) and \( t = 1 \) gives \( i = 3 \). Thus, we get \( \text{ECT}_\varepsilon(\xi, -0.3) = 3 \) and \( \text{ECT}_\varepsilon(\xi, 1) = 2 \). The evaluation of the Radon transform is similar. First, we search for indices \( i \) such that \( T_{c}[i] = t \). In that specific case, the method outputs \( \mathcal{R}_\varepsilon(\xi, t) = E_{c}[i] \).

Apart from this, the computations are identical.

<table>
<thead>
<tr>
<th>( \varepsilon )</th>
<th>( \text{Crit}^\text{ord}(\varepsilon) )</th>
<th>( \Delta^\text{ord}_\varepsilon(\varepsilon) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>((-1, -1))</td>
<td>([-28, 16, 44, 34])</td>
<td>([-1, -1, -1, -1])</td>
</tr>
<tr>
<td>((-1, 1))</td>
<td>([-42, 18, 32, 34])</td>
<td>([-1, 1, -1, 1])</td>
</tr>
<tr>
<td>((1, -1))</td>
<td>([-42, 18, 32, 34])</td>
<td>([0, 42])</td>
</tr>
<tr>
<td>((1, 1))</td>
<td>([-28, 16, 44, 34])</td>
<td>([0, 28, 16, 30])</td>
</tr>
</tbody>
</table>
A.4 Vectorization

The vectorization routine of the ECT (similar to the vectorization of the Radon transform) takes as input two bounds $a$ and $b$, and a number $N$ of points. Then, it returns $\text{ECT}_{\varphi}(\xi, a + \frac{i(b-a)}{N})$ for all $i \in [0, \ldots, N-1]$. For instance, if $a = -2.0$, $b = 2.0$, and $N = 5$, we first iterate over $i = 0, \ldots, 4$. For $i = 0$, we have $t_0 = a = -2$, so $\text{ECT}_{\varphi}(\xi, t_0) = E[1] = 1$. For $i = 1$, we know that $t_1 = -1 > t_0$, so we can start to search at index 1 in $T$. Then, we find that $\text{ECT}_{\varphi}(\xi, t_1) = E[1] = 1$. Finally, the values of $\text{ECT}_{\varphi}(\xi, t_0)$ for $t_0 \in [-2, -1, 0, 1, 2]$ are $[1, 1, 2, 2, 2]$.

B Proofs of mathematical results

In this section, we prove Lemmas 2 and 3 on which our algorithms are built. We first state and prove some necessary lemmas before giving the proofs of our main results. Throughout the section, let $\mathcal{P}$ be a polytopal complex in $\mathbb{R}^n$ and let $\xi : \mathbb{R}^n \to \mathbb{R}$ be a linear form that is generic on $\mathcal{P}$.

These results and notions introduced in Sections 2 and 3 are more naturally phrased and proven using Euler calculus [13, 17], that is, the integral calculus of constructible functions with respect to the Euler characteristic. For instance, our definition of Euler characteristic, Euler calculus formulation of our notions of constructible functions and of Euler integration in the core of the paper. Along the section, we introduce notions of Euler calculus only when necessary to keep the text accessible to the broadest audience and refer to [13] for more details.

For any $P \in \mathcal{P}$ we denote by $\mathbf{1}_P : \mathcal{P} \to \mathbb{Z}$ the weighted cubical complex with value 1 on $P$ and 0 on any over cells (even on proper faces of $P$). We call $\mathbf{1}_P$ the indicating function of $P$. Then, any weighted cubical complex $\varphi : \mathcal{P} \to \mathbb{Z}$ can be written as a sum $\varphi = \sum_{P \in \mathcal{P}} \varphi(P) \mathbf{1}_P$.

The following lemma gives an explicit expression of the Radon transform of indicating functions. Before stating it, let us make the following remark. The Euler characteristic, Euler characteristic transforms, Radon transforms and hybrid transforms of a function $\varphi : \mathcal{P} \to \mathbb{Z}$ and of its restriction to the complex $\mathcal{P}^\varphi$ induced by the collection $\{P \in \mathcal{P} : \varphi(P) \neq 0\}$ are equal.

> **Lemma 5.** Let $P \in \mathcal{P}$ and assume that $P$ is not a vertex of $\mathcal{P}$. Then, one has $R_n(\xi, \cdot) = (-1)^{\dim(P)-1} \mathbf{1}_{(\min_P f, \max_P f)}$.

**Proof.** This fact is an easy consequence of the Euler calculus formulation of our notions detailed in the beginning of this section. We give an elementary proof for completeness. Let us denote $\varphi = \mathbf{1}_P$. Since $\xi$ is linear, the set $P \cap \{\xi = t\}$ is a polytope (possibly empty) for any $t \in \mathbb{R}$. Moreover, since $\xi$ is generic on $\mathcal{P}$ and $P$ is not a vertex, this polytope is either a vertex of $P$ or it has dimension $\dim(P) - 1$.

Let $t \in \mathbb{R}$. We distinguish three cases. First assume that $t \notin [\min_P \xi, \max_P \xi]$. Since $P$ is connected and $\xi$ is continuous, one has that $[\min_P \xi, \max_P \xi] = \{\xi(P)\}$. Therefore, the polytope $P \cap \{\xi = t\}$ is empty, so is $\mathcal{P}^\varphi_{\{\xi = t\}}$ and hence $R_n(\xi, t) = 0$.
Moreover, similarly, as a consequence, one has whenever they are defined. Moreover, following the notations of the beginning of this section, the continuous map min = R\ is non-empty. Moreover, the polytope P = ξ \in \{ξ = t\} is neither empty nor a vertex. Therefore, this polytope has dimension dim(P) − 1. Since Pφ consists of P and all its faces, the polytopal complex Pφ_{ξ=t} contains only one polytope of dimension dim(P) − 1 which is Q. Any other polytope Q’ of Pφ_{ξ=t} has dimension smaller or equal to dim(P) − 2. In such a case, one has φ_{ξ=t}(Q') ≠ φ(P) and thus φ_{ξ=t}(Q') = 0. As a consequence, one has Rφ(ξ, t) = φ_{ξ=t}(Q)(−1)^{dim(Q)} = (−1)^{dim(P)−1}. Hence the result.

For any vertex x of P, we denote by LwSt^∗(ξ, x) the set LwSt(ξ, x) \ {x}. Similarly, we define the set UpSt^∗(ξ, x). Moreover, for any function h : R → R we denote one-sided limits at any t ∈ R by:

\[ h(t−) = \lim \limits_{s→t} h(s) \quad \text{and} \quad h(t+) = \lim \limits_{s→t} h(s), \]

whenever they are defined. Moreover, following the notations of the beginning of this section, Euler calculus ensures that for any φ : P → Z the function Rφ(ξ, ⋅) = ξ, φ is a finite sum of indicating functions of real intervals [13, Thm. 2.3.(i)], hence admits all one-sided limits both from above and from below. The following lemma provides additional details on the Radon transform of indicating functions. It will be crucial in our study of critical points (Lemma 7).

**Lemma 6.** Let x ∈ Vert(P) be such that ⟨ξ; x⟩ = t and let P ∈ St(x). One has

\[
R_{1_P}(ξ, t) = \begin{cases} 
1 & (P = \{x\}), \\
0 & (P ∈ LwSt^∗(ξ, x) \cup UpSt^∗(ξ, x)), \\
(−1)^{dim(P)−1} & (P \not∈ LwSt(ξ, x) \cup UpSt(ξ, x)). 
\end{cases}
\]

Moreover,

\[
R_{1_P}(ξ, t−) = \begin{cases} 
0 & (P ∈ UpSt(ξ, x)), \\
(−1)^{dim(P)−1} & (P \not∈ UpSt(ξ, x)). 
\end{cases}
\]

Similarly,

\[
R_{1_P}(ξ, t+) = \begin{cases} 
0 & (P ∈ LwSt(ξ, x)), \\
(−1)^{dim(P)−1} & (P \not∈ LwSt(ξ, x)). 
\end{cases}
\]

**Proof.** Throughout the proof, denote φ = 1_P. Let us first prove the first equality. The case P = \{x\} is clear. If P ∈ LwSt^∗(x) \cup UpSt^∗(x), then Pφ_{ξ=t} = \{x\} and Rφ(ξ, t) = φ(x) = 0. Now, suppose that P \not∈ LwSt(ξ, x) \cup UpSt(ξ, x). Since t ∈ ξ(P), the polytope Q = P \cap \{ξ = t\} is non-empty. Moreover, the polytope Q is not a vertex, for otherwise the generic linear form ξ would be constant on the edge between the distinct vertices Q and x. Therefore, the polytope Q has dimension dim(P) − 1. As in the proof Lemma 5, one can thus show that Rφ(ξ, t) = φ_{ξ=t}(Q)(−1)^{dim(Q)} = (−1)^{dim(P)−1}.

Now, let us prove the second equality. If P ∈ UpSt(ξ, x), then P ⊆ \{ξ ≥ t\} and \{ξ = s\} ∩ P = ∅ for all s < t. Hence Rφ(ξ, t−) = 0. If P ∈ St(x) \ UpSt(ξ, x), then min_P ξ < t. Since both min_P ξ and t lie in the image of the connected set P by the continuous map ξ, then [min_P ξ, t] ⊆ ξ(P). Therefore, for any s ∈ [min_P ξ, t], the polytope
$Q_s = \{ \xi = s \} \cap P$ is non-empty. The same argument as in the previous paragraph shows that $Q_s$ is not a vertex and hence has dimension $\dim(P) - 1$. Once again, we can conclude that $R_\varphi(t, t) = \varphi(\xi = t)(Q_s)(-1)^{\dim(Q_s)} = (-1)^{\dim(P) - 1}$.

The proof of the last equality follows from the last one applied to the linear form $-\xi$. ▶

**Lemma 7.** Let $\varphi : P \rightarrow \mathbb{Z}$. For any $t \in \mathbb{R}$,

$$\mathcal{R}_\varphi(\xi, t) - \mathcal{R}_\varphi(\xi, t^-) = \sum_{\xi \in \text{Crit}_t(\xi)} \Delta_\varphi(\xi, x),$$

$$\mathcal{R}_\varphi(\xi, t^+) - \mathcal{R}_\varphi(\xi, t^-) = \sum_{\xi \in \text{Crit}_t^0(\xi)} \Delta_\varphi(\xi, x).$$

**Proof.** Let us prove the first equality. Let $t \in \mathbb{R}$. Since $\varphi = \sum_{P \in P} \varphi(P)1_p$ one has:

$$\mathcal{R}_\varphi(\xi, t) - \mathcal{R}_\varphi(\xi, t^-) = \sum_{P \in P} \varphi(P)(\mathcal{R}_1(\xi, t) - \mathcal{R}_1(\xi, t^-)). \quad (B.1)$$

Fix $P \in P$. Suppose that no vertex of $P$ belongs to the level set $\{ \xi = t \}$. Since $\xi$ is linear, it implies that $t \neq \min f$ and $t \neq \max f$. Thus $t \in (\min f, \max f)$ or $t \in \mathbb{R} \setminus [\min f, \max f]$ and Lemma 5 implies that:

$$\mathcal{R}_1(\xi, t) - \mathcal{R}_1(\xi, t^-) = (-1)^{\dim(P) - 1}(1_{\min f, \max f}(\xi, t) - 1_{\min f, \max f}(\xi, t^-)) = 0.$$ 

Now, suppose there exists a vertex $x$ of $P$ such that $\langle \xi; x \rangle = t$. Lemma 6 implies that if $P \notin \text{LwSt}(\xi, x)$, then $\mathcal{R}_1(\xi, t) = \mathcal{R}_1(\xi, t^-) = 0$. Indeed, either $P \in \text{UpSt}(\xi, x)$ and then $\mathcal{R}_1(\xi, t) = \mathcal{R}_1(\xi, t^-) = 0$, or $P \notin \text{LwSt}(\xi, x) \cup \text{UpSt}(\xi, x)$ and then $\mathcal{R}_1(\xi, t) = \mathcal{R}_1(\xi, t^-) = (-1)^{\dim(P) - 1}$. Therefore, (B.1) can be written:

$$\mathcal{R}_\varphi(\xi, t) - \mathcal{R}_\varphi(\xi, t^-) = \sum_{P \in P} \varphi(P)(\mathcal{R}_1(\xi, t) - \mathcal{R}_1(\xi, t^-)) \quad (B.2)$$

where the sum in the right-hand side is over all polytopes $P \in P$ such that there exists $x \in \text{Vert}(P)$ with $\{ \xi = t \}$ and $P \in \text{LwSt}(\xi, x)$. Moreover, for any $P \in P$ there is at most one vertex $x$ of $P$ such that $P \in \text{LwSt}(\xi, x)$. Indeed, if there were $x \neq x'$ such that $P \in \text{LwSt}(\xi, x) \cap \text{LwSt}(\xi, x')$, then $\langle \xi; x \rangle = \max f = \langle \xi; x' \rangle$. Denote $m = \langle \xi; x \rangle = \langle \xi; x' \rangle$. Then $\{ \xi = m \} \cap P$ is a face of $P$ that is not a vertex and on which the generic linear form $\xi$ is constant, a contradiction.

Therefore, (B.2) can be written:

$$\mathcal{R}_\varphi(\xi, t) - \mathcal{R}_\varphi(\xi, t^-) = \sum_{x \in \text{Vert}(P)} \sum_{P \in \text{LwSt}(\xi, x)} \varphi(P) \left( \mathcal{R}_\varphi(\xi, t) - \mathcal{R}_\varphi(\xi, t^-) \right). \quad (B.3)$$

Let $x \in \text{Vert}(P)$ such that $\langle \xi; x \rangle = t$ and $P \in \text{LwSt}(\xi, x)$. Lemma 6 implies that $\mathcal{R}_1(\xi, t) = 1$ if $P = \{ x \}$ and 0 if $P \in \text{LwSt}(\xi, x)$. The same lemma implies that $\mathcal{R}_1(\xi, t^-) = 0$ if $P = \{ x \}$ and $(-1)^{\dim(P) - 1}$ if $P \in \text{LwSt}(\xi, x)$. Therefore, one has:

$$\sum_{P \in \text{LwSt}(\xi, x)} \varphi(P) \left( \mathcal{R}_1(\xi, t) - \mathcal{R}_1(\xi, t^-) \right) = \varphi(x) - \sum_{P \in \text{LwSt}(\xi, x)} (-1)^{\dim(P) - 1} \varphi(P) = \sum_{P \in \text{LwSt}(\xi, x)} (-1)^{\dim(P)} \varphi(P) = \Delta_\varphi(\xi, x).$$
Therefore, (B.3) can be written:

\[ R_\phi(\xi, t) - R_\phi(\xi, t^-) = \sum_{x \in \text{Vert}(P) \atop \langle \xi, x \rangle = t} \Delta^-_\phi(\xi, x). \]  

(B.4)

The set \( \text{Crit}^-_\phi(\xi) \) being the set of vertices \( x \) with non-zero \( \Delta^-_\phi(\xi, x) \), the first equality of the lemma follows.

Now, let us prove the second equality. Replacing \( \xi \) by \( -\xi \), one gets from (B.4) that:

\[ R_\phi(\xi, t^+) - R_\phi(\xi, t) = \sum_{x \in \text{Vert}(P) \atop \langle \xi, x \rangle = t} \chi \left( \varphi \mid \text{UpSt}(\xi, x) \right). \]  

(B.5)

Hence the result, using (B.4), (B.5), and the fact that \( R_\phi(\xi, t^+) - R_\phi(\xi, t^-) = R_\phi(\xi, t^+) - R_\phi(\xi, t) - (R_\phi(\xi, t) - R_\phi(\xi, t^-)). \)

\[ \square \]

**B.1 Proof of Lemma 2**

Let us prove the equality for the Radon transform. Euler calculus [13, Thm. 2.3.(i)] ensures that \( \xi, \hat{\phi} \) is a finite sum of indicating functions of real intervals. Therefore, it is locally constant on the complement of a finite number of points in \( \mathbb{R} \). It is then an easy exercise to check that Lemma 7 yields the result:

\[ R_\phi(\xi, \cdot) = \sum_{x \in \text{Crit}^-_\phi(\xi)} \Delta^-_\phi(\xi, x) \cdot 1_{\{\langle \xi, x \rangle\}} + \sum_{x \in \text{Crit}^{\text{ord}}_\phi(\xi)} \Delta^{\text{ord}}_\phi(\xi, x) \cdot 1_{\{\langle \xi, x \rangle, +\infty\}}. \]  

(B.6)

By definition of the hybrid transform, integrating (B.6) against a locally integrable kernel \( \kappa : \mathbb{R} \to \mathbb{C} \) yields the result for \( \text{HT}_\phi(\xi) \).

The formula for the ECT can be easily shown using the classical convolution operation of Euler calculus [13, Sec. 4]. This bilinear operator takes two functions \( \theta \) and \( \theta' \) which are finite sums of indicating functions of intervals of \( \mathbb{R} \) and returns a function \( \theta \ast \theta' \) with the same property. One can show that \( \text{ECT}_\phi(\xi, t) = R_\phi(\xi, \cdot) \ast 1_{[0, +\infty)}(t) \); see for instance [11, Ex. 7.4, Prop. 7.5]. The expression for the ECT follows then from (B.6) and the fact that for \( a \in \mathbb{R} \) one has \( 1_{(a, +\infty)} \ast 1_{[0, +\infty)} = 0 \) and \( 1_a \ast 1_{[0, +\infty)} = 1_{[a, +\infty)} \); see [13, Exs. 4.1, 4.3]. This concludes the proof of Lemma 2.

\[ \square \]

**B.2 Proof of Lemma 3**

Let \( \mathcal{C} \) be a cubical complex. In that case, any edge \{\( x, y \}\} \in \mathcal{C} is such that \( x - y \) is colinear to a canonical basis vector of \( \mathbb{R}^n \). Let \( \xi \) and \( \xi' \) be in \( \mathbb{R}^n \setminus \{0\} \) with \( \xi = \xi' \). Since \( \xi \) and \( \xi' \) have same sign vector, we have \( \text{sgn}(\xi; x - y) = \text{sgn}(\xi'; x - y) \) for all edge \{\( x, y \}\} \in \mathcal{C}. It is an easy exercise to check that thus \( \xi \) and \( \xi' \) induce the same order on \( \text{Vert}(\mathcal{C}) \). As a consequence, the lower and upper stars of vertices of \( \mathcal{C} \) with respect to \( \xi \) and \( \xi' \) are the same. Moreover, the sets and numbers \( \Delta^-_\phi(\xi, x) \), \( \Delta^{\text{ord}}_\phi(\xi, x) \), \( \text{Crit}^-_\phi(\xi) \) and \( \text{Crit}^{\text{ord}}_\phi(\xi) \) depend only on \( \text{LwSt}(\xi, x) \) and \( \text{UpSt}(\xi, x) \) (and not on \( \xi \)), hence the result.

\[ \square \]
Engineering A* Search for the Flip Distance of Plane Triangulations

Philip Mayer
Institute of Computer Science, University of Bonn, Germany

Petra Mutzel
Institute of Computer Science, University of Bonn, Germany

Abstract
The flip distance for two triangulations of a point set is defined as the smallest number of edge flips needed to transform one triangulation into another, where an edge flip is the act of replacing an edge of a triangulation by a different edge such that the result remains a triangulation.

We adapt and engineer a sophisticated A* search algorithm acting on the so-called flip graph. In particular, we prove that previously proposed lower bounds for the flip distance form consistent heuristics for A* and show that they can be computed efficiently using dynamic algorithms. As an alternative approach, we present an integer linear program (ILP) for the flip distance problem.

We experimentally evaluate our approaches on a new real-world benchmark data set based on an application in geodesy, namely sea surface reconstruction. Our evaluation reveals that A* search consistently outperforms our ILP formulation as well as a naive baseline, which is bidirectional breadth-first search. In particular, the runtime of our approach improves upon the baseline by more than two orders of magnitude. Furthermore, our A* search successfully solves most of the considered sea surface instances with up to 41 points. This is a substantial improvement compared to the baseline, which struggles with subsets of the real-world data of size 25.

Lastly, to allow the consideration of global sea level data, we developed a decomposition-based heuristic for the flip distance. In our experiments it yields optimal flip distance values for most of the considered sea level data and it can be applied to large data sets due to its fast runtime.

1 Introduction
A triangulation $D$ of a point set $S \subseteq \mathbb{R}^2$ is a maximal set of non-intersecting straight line edges with endpoints in $S$. Triangulations play an important role in theoretical as well as applied computational geometry. Applications can, for example, be found in computer graphics [15], finite element mesh generation [33], or reconstruction problems in geodesy [28]. Usually, the set $\mathcal{T}(S)$ of all triangulations of a point set $S$ is exponential in size.

One way to navigate the set $\mathcal{T}(S)$ is by edge or diagonal flips. Let $e$ be a diagonal of $D$, i.e., an edge that is not part of the convex hull, and let $Q$ be the quadrilateral given by the two adjacent triangles of $e$. If $Q$ is convex, then the edge flip of $e$ in $D$ is given by substituting the diagonal $e$ by the other diagonal $f$ in $Q$; see Figure 1. This procedure of

© Philip Mayer and Petra Mutzel; licensed under Creative Commons License CC-BY 4.0
22nd International Symposium on Experimental Algorithms (SEA 2024).
Editor: Leo Liberti; Article No. 23; pp. 23:1–23:20
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
flipping diagonals is often used for iterative local optimization of triangulations with respect to some criteria. The most famous application of this idea is the construction of the Delaunay triangulation [8], which uses a min–max angle criterion [20].

A more general question regarding the flip distance is the following: For two triangulations $D$ and $D'$, what is the minimal number of edge flips needed to transform $D$ into $D'$? This number is called the flip distance $d_F(D, D')$ of $D$ and $D'$. The set $\mathcal{T}(S)$ of all triangulations and the flip distance $d_F(o, o)$ form a metric space, since the flip distance is symmetric, non-negative, and satisfies the triangle inequality.

Another interesting property of the flip operation is that it induces a graph structure on $\mathcal{T}(S)$, which is called the flip graph $\mathcal{FG}(S)$ of $S$. The set of vertices of the graph corresponds to $\mathcal{T}(S)$ and two triangulations $D, D' \in \mathcal{T}(S)$ are connected by an (unweighted) edge if one can be transformed into the other by exactly one flip. Since every triangulation can be transformed to the Delaunay triangulation in a finite number of flips [20], the flip graph of a point set is connected. Additionally, the flip distance $d_F(D, D')$ corresponds to the shortest path between $D$ and $D'$ in $\mathcal{FG}(S)$. Thus, standard methods for the calculation of shortest paths can be used to compute the flip distance. It should be noted that the flip graph can have an exponential number of vertices and edges, which implies an exponential runtime for the shortest path algorithms. In fact, Pilz [30] showed that deciding if the flip distance of a pair of triangulations is at most $k$ is NP-complete.

The motivation for our work is a recent paper by Nitzke, Niedermann, Fenoglio-Marc, Kusche and Haunert [28] who used optimal minimum error triangulations to reconstruct the sea surface using tide gauge stations and satellite data. In their work, they compute an optimized triangulation for every month in a period of 22 years and use these optimized triangulations to reconstruct the sea surface at other points in time. Due to seasonal changes in sea level, it is usually more accurate to reconstruct the sea surface for a given date (month and year) from an optimized triangulation for the same month (but a different year). However, due to local shifts and fluctuations, this expectation does not always materialize. Therefore, to gain more insights which triangulations should be used for the reconstruction at a specific point in time, it is of interest to cluster the set of optimized triangulations with respect to some distance measure. One candidate for a distance measure is the flip distance since it is a natural way to define the distance between two triangulations. Additionally, $d_F(o, o)$ is a metric on $\mathcal{T}(S)$, which allows the use of most generic clustering algorithms.

The literature on theoretical results regarding the flip distance of point sets is extensive, but to the best of our knowledge, the practical computation of the flip distance for real-world applications has yet to be discussed in any detail. With our work, we want to initiate this discussion.
Our contribution

- We adapt and engineer an $A^*$ search algorithm for the exact computation of the flip distance. For this, we prove that the lower bound given by Eppstein [9] constitutes a consistent heuristic in the context of the $A^*$ paradigm. For the calculation of the heuristic values during a node extension, we use dynamic algorithms to improve the naive runtime by a factor of $O(n)$.
- As an alternative approach, we suggest the first integer linear program for the flip distance problem.
- We present a new decomposition-based upper bound for the flip distance problem that can be effectively applied to the real-world data.
- We propose and discuss a new benchmark data set for the flip distance problem. The data set consists of real-world data based on the sea surface reconstruction problem as well as randomly generated data for systematic testing.
- We use our benchmark data to perform an experimental evaluation of our algorithms against bidirectional breadth-first search as a baseline. Our analysis focuses on the runtime and highlights the impact of employing good heuristics on the algorithms’ performance.
- Finally, we discuss the performance of our decomposition heuristic on the real-world data.

The remaining paper is structured as follows: We start in Section 2 by providing an overview of related work on the flip distance problem. Then, in Section 3, we present our $A^*$ algorithm. Next, in Section 4, we summarize the known lower bounds and show that they are consistent heuristics for $A^*$. Additionally, we show how to compute the heuristics efficiently during the $A^*$ search. We present our integer linear program (ILP) for the flip distance problem in Section 5 and our decomposition heuristic in Section 6. In Section 7 we discuss the benchmark data set and conduct our experimental analysis of the algorithms. Lastly, we give our conclusion and propose further research directions in Section 8.

2 Related Works

For a more detailed survey of known results, we refer to the work by Bose and Hurtado [3].

Convex polygons/points in convex position. The setting with $n$ points in convex position has been studied extensively. The main reason for this is that there is a bijection between triangulations of the $n$-gon and binary trees with $n - 2$ inner nodes. Additionally, diagonal flips in triangulations correspond to rotations in these trees. This is, for example, proven by Sleator, Tarjan and Thurston [32]. In the same work, they show that $2n - 10$ for $n > 12$ is a tight upper bound on the flip distance in the convex case. Baril [2] and Pallo [29] proposed upper and lower bounds on the flip distance and experimentally evaluated them.

There are multiple fixed-parameter tractable (FPT) algorithms for the convex flip distance where the parameter $k$ is the flip distance [6, 25, 21]. The best known FPT algorithm with runtime $O(3.82^k)$ was developed by Li and Xia [21].

Interestingly, the complexity of the flip distance problem remains unknown for convex polygons, despite the extensive amount of research invested into the problem.

In his bachelor thesis [22], Lipp used standard shortest path methods, i.e., BFS and $A^*$, to compute the convex flip distance. To the best of our knowledge, this is the only work that implements and evaluates a method to compute the flip distance to optimality.
Points in arbitrary position. The first published result for the flip distance problem in the general case was the work by Lawson [19], which showed that any two triangulations of a point set can be transformed into each other by a sequence of flips. Additionally, he showed that $O(n^2)$ flips are sufficient [19]. Hurtado, Noy and Urrutia [16] showed that this upper bound is tight by providing example triangulations for which $\Omega(n^2)$ flips are needed to transform one triangulation into the other. In a secondary work [20] Lawson also showed that any triangulation can be transformed into the well-known Delaunay triangulation [8].

The work [13] by Hanke, Ottmann and Schuierer shows that for two triangulations the number of (proper) intersections of edges provides an upper bound on the flip distance. Eppstein [9] proved that for point sets that do not contain empty pentagons the flip distance can be computed in $O(n^2)$ time. As a byproduct of his discussion, he derives a lower bound on the flip distance for point sets that contain empty pentagons, which we discuss in Section 4.

A first FPT algorithm with runtime $O^*(k \cdot c^k)$, where $c \leq 2 \cdot 14^{11}$, was given by Kanj, Sedgwick, and Xia [17]. Later, Feng, Li, Meng and Wang [10] improved the algorithm, which led to a runtime of $O^*(k \cdot 32^k)$.

Lubiw and Pathak [24] as well as Pilz [30] independently showed that the flip distance problem is NP-complete. Additionally, Pilz showed the APX-hardness of the problem.

3 Adapting A* Search for the Flip Graph

In this section, we present an A* search algorithm adapted for the flip graph $FG(S)$ of a point set $S$ of size $n$ with $c$ points on its convex hull. To avoid confusion with other graphs, which are defined in Section 4, we call the triangulations nodes of $FG(S)$ instead of vertices.

We start with some basic definitions (contextualized in the triangulation setting). A heuristic with respect to a target triangulation $D_t$ is a function $h^{D_t}: T(S) \rightarrow \mathbb{R}$. We say $h^{D_t}$ is admissible if for all triangulations $D$ the heuristic $h^{D_t}(D)$ is a lower bound for $d_F(D, D_t)$. The heuristic is consistent if $h^{D_t}(D_t) = 0$ and for all pairs of adjacent triangulations $D$ and $D'$ in the flip graph $FG(S)$ we have $h^{D_t}(D) \leq 1 + h^{D_t}(D')$ and vice versa. From now on we assume that the target triangulation is fixed and we write $h(D)$ instead of $h^{D_t}(D)$.

The A* search algorithm first proposed by Hart, Nilsson and Raphael [14] keeps a priority queue $Q$ of open nodes that need to be processed. It starts with $Q = \{D_0\}$ and in every iteration the node $D$ from $Q$ with minimal value $f(D) = d_F(D_0, D) + h(D)$ is extended. During the extension of a node $D$, it is removed from $Q$, marked as closed and all of its neighbors $D'$ are opened by adding them to $Q$ with their respective $f(D')$ values. The search is complete when $D_t$ is closed. If $h$ is admissible the resulting path is optimal. Additionally, neighbors that are marked as closed do not need to be re-opened during a node extension if $h$ is consistent. Note that the heuristics we present in Section 4 are consistent.

Next, we discuss the data structures that are needed to traverse $FG(S)$. Different from usual A* algorithms, the graph $FG(S)$ is not known in its entirety and would be far too large to be stored in memory, even for small instances. Thus, we build the graph $FG(S)$ on the fly and only add a node to the graph when it is first opened. Since every node is associated with a triangulation $D$, we need to represent $D$ in some way. For the sake of simplicity, we assign an integer value to every triangle that can occur in a triangulation. Note that the number of triangles in a triangulation of $n$ points with $c$ points on the convex hull is given by $z = 2n - c - 2$; see for example [7]. Hence, a triangulation can be represented by a sorted

\footnote{Our definition differs from the usual definition, but they are equivalent since our graph is unweighted.}
Algorithm 1 AStarSearch($D_s, D_t$).

1: $Q ←$ PriorityQueue, OPEN ← HashMap, CLOSED ← HashMap
2: $D_s.d = 0$
3: $D_s.f = \text{computeHeuristic}(D_s, D_t)$
4: $Q$.insert($D_s, D_s.f$), OPEN.add($D_s$)
5: \textbf{while} $Q \neq \emptyset$ \textbf{do}
6: \hspace{1em} $D = Q$.extractMin()
7: \hspace{1em} OPEN.remove($D$), CLOSED.add($D$)
8: \hspace{1em} \textbf{if} $D = D_t$ \textbf{then return} $D.d$
9: \hspace{1em} $\text{computeHeuristic}(D, D_t)$ \hspace{1em} \textgreater{} only executed for the computation of $h_E$; see Section 4
10: \hspace{1em} \textbf{for} each valid flip $e$ of $D$ \textbf{do}
11: \hspace{2em} $D′ = \text{doFlip}(D, e)$
12: \hspace{2em} \textbf{if} $D′ \in$ CLOSED \textbf{then continue}
13: \hspace{2em} $h = \text{computeHeuristicNeighbor}(D′, D, D_t)$
14: \hspace{2em} $D′.d = D.d + 1$
15: \hspace{2em} $D′.f = D′.d + h$
16: \hspace{2em} \textbf{if} $D′ \in$ OPEN \textbf{then}
17: \hspace{3em} $Q$.decreaseKey($D′$, $D′.f$) \hspace{1em} \textgreater{} only executed if $D′.f$ is smaller than the known value
18: \hspace{2em} \textbf{else}
19: \hspace{3em} $Q$.insert($D′, D′.f$), OPEN.add($D′$)

array of $z$ integer values. This representation allows for easy equality tests and we can use it to derive all diagonals and their corresponding quadrilaterals/flip in linear time. We keep track of nodes that have been opened during the search by storing them in a hashmap using the triangulation-representation as keys. The value of a triangulation $D$ in the hashmap is given by the tuple $(h(D), d_F(D_s, D), i)$ where $i$ is the position in the priority queue.

Algorithm 1 summarizes our $A^*$ algorithm. As stated before, in addition to the usual priority queue $Q$ we need to keep track of all opened and closed triangulations, which is done in OPEN and CLOSED. Closed nodes can be skipped in line 12, because our used heuristics are consistent. Finally, we note that the calculation of the heuristic value for a triangulation $D′$ in line 13 depends on its parent node $D$. This is the case because we want to dynamically update the heuristics using the information of the parent node. The dynamic updates for the different heuristics are discussed in Section 4.

4 Lower Bounds on the Flip Distance

In this section, we present two heuristics given by known lower bounds for the flip distance and show how to compute them efficiently during the $A^*$ search.

A simple lower bound. Let $D$ and $D′$ be two triangulations in $T(S)$. Then we can define $l_S(D, D′)$ to be the number of diagonals that are in $D$ and not in $D′$. Every diagonal in $D$ that does not coincide with a diagonal in $D′$ has to be flipped at some point. Thus, $l_S(D, D′)$ is a lower bound on the flip distance. In Figure 2 (a) the lower bound $l_S(D, D′)$ is five. Moreover, adjacent triangulations $D$ and $D$ in FG(S) only differ by one diagonal, which implies $|l_S(D, D′) − l_S(D, D′)| \leq 1$. Consequently, we get the following simple observation:

\textbf{Observation 1.} The heuristic $h_S(D) = l_S(D, D_t)$ is admissible and consistent.

A refined lower bound by Eppstein. To describe Eppstein’s [9] heuristic, we first need an additional graph, which we call the quadrilateral graph $QG$. The vertex set of $QG$ is given by all $O(n^2)$ diagonals that may be used in a triangulation of $S$. Two diagonals $e = \overline{uv}$ and
Search for the Flip Distance of Plane Triangulations

In (a) two triangulations $D$ (red) and $D'$ (dashed blue) of a point set are given. Diagonals that coincide are indicated in black. In (b) the QG graph is depicted, the relevant diagonals are indicated in red and blue. A minimal flip sequence is given by $ab \rightarrow df, af \rightarrow cd, de \rightarrow bh, bd \rightarrow fh, df \rightarrow ch, eh \rightarrow bg, bh \rightarrow fg$. The paths corresponding to the flip sequence are given as wavy edges and the shortest paths corresponding to a minimum weight perfect matching in $B(D, D')$ are the fat black edges. We have $l_S(D, D') = 5$, $l_E(D, D') = 6$ and $d_F(D, D') = 7$.

e' = uv' are connected by an edge in QG if $e$ and $e'$ properly intersect and the quadrilateral $q$ given by $\{u, u', v, v'\}$ does not contain any points of $S$ in its interior. Note that $q$ is a convex quadrilateral if $e$ and $e'$ properly intersect. An example of a QG graph is depicted in Figure 2.

Figure 2 In (a) two triangulations $D$ and $D'$ of a point set are given. Diagonals that coincide are indicated in black. In (b) the QG graph is depicted, the relevant diagonals are indicated in red and blue. A minimal flip sequence is given by $ab \rightarrow df, af \rightarrow cd, de \rightarrow bh, bd \rightarrow fh, df \rightarrow ch, eh \rightarrow bg, bh \rightarrow fg$. The paths corresponding to the flip sequence are given as wavy edges and the shortest paths corresponding to a minimum weight perfect matching in $B(D, D')$ are the fat black edges. We have $l_S(D, D') = 5$, $l_E(D, D') = 6$ and $d_F(D, D') = 7$.

Observation 2 (Eppstein [9]). Let $D$ and $D'$ be two triangulations of $S$. Any sequence of flips that transforms $D$ to $D'$ corresponds to a set of walks in QG that connect diagonals of $D$ with diagonals of $D'$.

These walks are indicated for a minimal flip sequence in Figure 2. Now we can describe the lower bound $l_E(D, D')$ given by Eppstein: Let $B(D, D')$ be the complete bipartite graph with one part given by the diagonals of $D$ and the other part by the diagonals of $D'$. We define the cost $c(e, e')$ of an edge between a diagonal of $e \in D$ and a diagonal $e' \in D'$ to be the length of the shortest path between $e$ and $e'$ in QG. Then, $l_E(D, D')$ is given by the value of a minimum weight perfect matching (MWPM) on $B(D, D')$ with respect to the costs $c$.

Proposition 3. The heuristic $h_E(D) = l_E(D, D_t)$ is admissible and consistent.

Proof. The proof is given in Appendix A.

The only edges in the bipartite graph $B(D, D_t)$ that have value zero are edges between common diagonals. All other edges in the matching have at least value one, which implies:

Observation 4. For every possible triangulation $D$ of $S$ we have $h_E(D) \geq h_S(D)$.

Before we discuss how to compute these heuristics efficiently during the A* search, we investigate the runtime of a node extension, i.e., one execution of the while loop in Algorithm 1. Let $t[h(D')]$ be the runtime of one heuristic computation and $t[b(D')]$ be the runtime of the bookkeeping in the priority queue and hashmaps for one neighbor $D'$. It is well known [7] that every triangulation of a point set with $c$ points on the convex hull consists of $3n - c - 3$ edges. Thus, for one triangulation $O(n)$ diagonals are candidates for possible flips. After deriving all diagonals and their (valid) flips from the representation of $D$ in linear time, we
can perform each flip in constant time. Note that for every neighbor that is inserted into the hashmap a new representation array must be created, which implies that the processing time for a neighbor is at least $O(n)$. Thus, one node extension needs $O(n \cdot (n + t[h(D')] + t[b(D')]$ time.

**Extending nodes with the simple heuristic.** For the initial triangulation $D_s$, the heuristic value $h_S(D_s)$ has to be calculated from scratch. Note that we can represent the edge sets of $D_s$ and $D_t$ by integer arrays $A_s$ and $A_t$. After sorting the arrays, the number $k$ of elements that are present in both arrays can be computed using a simultaneous pass through both arrays. Thus, the heuristic, given by $h_S(D_s) = |A_s| - k$, can be computed in $O(n \log n)$ time.

If we extend a node $D$, we can query the heuristic value $h_S(D)$ of $D$. For all neighbors $D'$ we can now compute the heuristic value $h_S(D')$ in constant time. Let $e$ be the diagonal that is flipped to $e'$ in $D'$. Then we only need to check if $e \in D_t$ or $e' \in D_t$ and we get

$$h_S(D') = \begin{cases} h_S(D) + 1 & \text{if } e \in D_t \text{ and } e' \notin D_t \\ h_S(D) - 1 & \text{if } e \notin D_t \text{ and } e' \in D_t \\ h_S(D) & \text{otherwise.} \end{cases}$$

It follows that the processing time for a single neighbor is dominated by the construction of the representation and we get the runtime $O(n^2 + n \cdot t[b(D')])$ for a node extension with the simple heuristic.

**Extending nodes with Eppstein’s heuristic.** To allow for a fast construction of the cost matrix $C_{DD'}$ that corresponds to the bipartite graph $B(D, D')$, we compute the distance matrix $D_{QG}$ for the quadrilateral graph $QG$ in a pre-processing step. To this end we need to solve the all-pairs shortest-path problem on $QG$. Since the graph $QG$ is unweighted with $O(n^2)$ vertices and $O(n^4)$ edges, we can solve the problem in time $O(n^6)$ using breadth first search. This runtime seems excessive, but for the instances we consider in our experiments, this is fast enough.

It remains to be shown how to compute the minimum weight perfect matching. For this, we can use the Hungarian algorithm which was introduced by Kuhn [18] and later refined by Munkres [27]. Their algorithm works in $O(n)$ phases that each take $O(n^2)$ time, which leads to a runtime of $O(n^3)$. Using this algorithm and the pre-processed matrix $D_{QG}$, the heuristic calculation for one neighbor can be done in $O(n^5)$. Since we have $O(n)$ neighbors per node extension, the runtime for a node extension with the (static) Hungarian algorithm is $O(n^4 + n \cdot t[b(D')]$).

If we consider a neighbor $D'$ during a node extension, only one diagonal $e_k$ changes with respect to the parent node $D$. This implies that for the other diagonals, all costs remain the same. Thus, we can derive the cost matrix $C_{DD'}$ for a neighbor by taking the cost matrix $C_{DD}$ of the parent node and changing exactly the row $k$ of costs corresponding to the flipped edge. It seems wasteful to recompute the matching from scratch. In fact, there is a dynamic version of the Hungarian algorithm established by Mills-Tettey, Stent, and Dias [26]. This dynamic algorithm only needs to perform one additional phase of the Hungarian algorithm per changed row. Hence, it has runtime $O(n^2)$ if only one row is changed.

If we want to use this dynamic algorithm for the neighbors of a node during the node’s extension, we need to know the perfect matching of the parent node and the corresponding labeling of the Hungarian algorithm. Storing this information for every opened triangulation would drastically increase the memory consumption. Instead, we perform a “redundant”
(static) computation of the matching and labeling for the parent node, which is indicated in line 9 in Algorithm 1. We can then compute the heuristics dynamically for the neighbors using the labels and matching of the parent node. Thus, we get the runtime
\[
\mathcal{O}(n^3 + n \cdot n^2 + n \cdot t[b(D')]) = \mathcal{O}(n^3 + n \cdot t[b(D')])
\]
for the node extension with \(\mathcal{O}(n)\) neighbors. Note that at least asymptotically the additional computation of the heuristic does not matter.

### 5 A Layered Integer Linear Program

For many combinatorial optimization problems integer programming based approaches perform very well. To the best of our knowledge, no ILP for the flip distance has been proposed so far, so we introduce one in this section. Let \(D\) and \(D'\) be two triangulations of a point set \(S\) given by \(n\) points. The idea of the ILP is to build a layered formulation where every layer corresponds to a valid triangulation and two consecutive layers differ by exactly one edge. If we fix the first layer to \(D\) and the last layer to \(D'\) and the number of layers is sufficiently large, then the number of changes between layers is the flip distance. To ensure that the number of layers \(L\) is big enough, we choose \(L\) to be the upper bound given by Hanke et al. [13]. As mentioned in Section 4, the number of edges of the triangulation \(D\) (and \(D')\) is \(M = 3n - c - 3\), where \(c\) is the size of the convex hull of \(S\). Let \(\mathcal{E}\) be the set of all edges that can be utilized by a triangulation of the point set \(S\). Additionally, let \(E \subseteq \mathcal{E}\) and \(E' \subseteq \mathcal{E}\) be the sets of edges for \(D\) and \(D'\). For every \(e \in \mathcal{E}\) and every \(0 \leq i \leq L\) we have variables \(x^i_e\) and \(d^i_e\). The \(x^i_e\) variables are used to encode valid triangulations on each layer and the \(d^i_e\) variables encode the deletion of an edge between layers \(i\) and \(i + 1\). Using these variables, the ILP is given by:

\[
\begin{align*}
\text{minimize} & \quad \sum_{k \leq L} \sum_{e \in \mathcal{E}} d^k_e \\
& \quad x^i_e + x^{i+1}_e \leq 1 \quad \forall e, f \in \mathcal{E}, e \text{ crosses } f, i \leq L \quad (T1) \\
& \quad \sum_{e \in \mathcal{E}} x^i_e = M \quad \forall i \leq L \quad (T2) \\
& \quad x^i_e - x^{i+1}_e - d^i_e \leq 0 \quad \forall e \in \mathcal{E}, \forall i \leq L - 1 \quad (F1) \\
& \quad \sum_{e \in \mathcal{E}} d^i_e \leq 1 \quad \forall i \leq L - 1 \quad (F2) \\
& \quad \sum_{e \in \mathcal{E}} d^{i+1}_e \leq \sum_{e \in \mathcal{E}} d^i_e \quad \forall i \leq L - 1 \quad (F3) \\
& \quad x^0_e = 1 \quad \forall e \in E, \quad x^0_e = 0 \quad \forall e \notin E \quad (I1) \\
& \quad x^L_e = 1 \quad \forall e \in E', \quad x^L_e = 0 \quad \forall e \notin E' \quad (I2) \\
& \quad x^i_e \in \{0, 1\}, \quad d^i_e \in \{0, 1\} \quad \forall e \in \mathcal{E}, \forall i \leq L \quad (B)
\end{align*}
\]

Since our triangulations have \(M\) edges and a triangulation is a set of non-intersecting edges, the constraints \((T1)\) and \((T2)\) guarantee that on every layer the set of positive edge variables corresponds to a valid triangulation. The constraints \((F1)\) encode edge deletions from one layer to the next, i.e., if we have a variables \(x^i_e = 1\) and \(x^{i+1}_e = 0\), we must also have \(d^i_e = 1\). Constraints \((F2)\) ensure that only one edge is deleted from one layer to the next. Note that constraints \((F1)\) do not forbid edge insertions, but \((T2)\) implies that every insertion must correspond to a deletion. Hence, we can only have one edge insertion between layers.
In (a) and (b) the triangulations $D_s$ and $D_t$ are shown. In (c) the common edges are given in black and the dotted red and blue edges indicate the non-common edges. In (d) the 2-connected component $C$ that contains the convex hull is indicated in black and the remaining points are given in gray. The final decomposition is given in (e) where all edges that can be flipped are indicated by dotted lines. Note that some common edges can now be flipped.

Additionally, this added edge must be the other diagonal of the (convex) quadrilateral given by the incident triangles of the deleted edge, since otherwise (T1) would be violated. Hence, every $d_i = 1$ variable encodes a valid flip. Constraints (F3) are used to break symmetries. They force the flips to be done as early as possible. This is only relevant if the number of layers is not minimal and they are not needed for the correctness of the formulation. The constraints (I1) and (I2) set the first and last layer to $D$ and $D'$, respectively. Finally, the objective function counts the overall number of edge deletions, which is exactly the number of flips.

We also considered formulations that use triangles (similar to De Loera et al. [23]) and quadrilaterals as variables, but we do not present them here, since they are very similar to the presented ILP and performed far worse.

### 6 A Decomposition Approach

In this section, we discuss a decomposition-based approach to heuristically solve the flip distance problem. The basic idea of the approach is that, if two triangulations $D_s$ and $D_t$ already have a common edge, then we assume that it is not flipped in an optimal flip sequence. This assumption is true in the convex case [32] and has been used for multiple FPT-algorithms [4, 6, 25]. For arbitrary point sets the assumption is not always correct (see Pilz [30]), but we get an upper bound for $d_F(D_s, D_t)$ by only flipping non-common edges. We start this section by discussing our decomposition approach and then we introduce a class of instances (that is relevant in our application) for which the approach is advantageous.

If we fix edges that coincide in both triangulations, there can be paths of fixed edges that split the instance into sub-problems that can be solved individually. Note that the boundaries of the sub-problems are not necessarily convex, but our algorithms also work with non-convex boundaries. Using this simple insight we can present our decomposition scheme (see Figure 3 for an illustration of the different steps):

1. Compute the set $F$ of edges that are used in $D_s$ as well as $D_t$.
2. Compute the 2-connected components of $F$ and find the component $C$ that contains the convex hull.
3. Compute all faces $F_i$ of the geometric graph given by $C$.
4. Distribute the remaining points and the sub-triangulations to the faces $F_i$.
5. Solve the sub-problems individually and combine their solutions.
The algorithm is designed to only fix a small, necessary set of edges that yield a decomposition. To this end, we only consider the (non-trivial) 2-connected components, since only those decompose our problem into smaller sub-problems. Additionally, we decided to only use the 2-connected component that contains the convex hull. This guarantees that every point from \( S \setminus C \) is contained in exactly one sub-problem.

The other known upper bound by Hanke et al. [13], also never flips common edges. Thus, our heuristic is always at least as good as the one by Hanke, as, unlike Hanke’s approach, it computes optimal sequences on the sub-problems given by non-coinciding edges. For arbitrary inputs, we cannot expect the decomposition approach to be fast, since we cannot assume that a lot of edges coincide and in particular, they may not form paths that split our instance into smaller sub-problems. We now propose a class of instances for which the decomposition approach can be used effectively.

**Higher-order Delaunay triangulations** The instances we are interested in are those where \( D_s \) and \( D_t \) both are higher-order Delaunay triangulations with order \( k \). Higher-order Delaunay triangulations [11, 31] are similar to the Delaunay triangulation while leaving room for optimization. More specifically, for every triangle \( T \) of a \( k \)-order Delaunay (\( k \)-OD) triangulation the circumcircle of \( T \) contains at most \( k \) points of \( S \). They are often used for interpolation tasks and in particular, they are used in the sea level reconstruction data. It is a well-known fact [11] that only \( O(nk) \) of the \( O(n^2) \) possible edges are used in all \( k \)-OD triangulations of a point set. For small \( k \) this results in a lot of fixed edges that are part of every \( k \)-OD triangulation. Figure 4 depicts the fixed edges of a point set for different orders \( k \). For orders \( k \leq 7 \) the experiments by Arutyunova et al. [1] show that, even for larger data sets, the \( k \)-OD fixed edges already yield a good decomposition. Thus, if \( D_s \) and \( D_t \) both have order \( k \leq 7 \), their common edges include the \( k \)-OD fixed edges and we can expect a useful decomposition of our instance.

### 7 Experiments

We start this section by presenting the used benchmark instances. Then we give some details on our implementation and evaluate our algorithms.

#### 7.1 Data

We use two types of instances: randomly generated instances and instances given by the sea surface reconstruction task introduced by Nitzke et al. [28].
Table 1 Overview of the relevant data and the number of non-trivial instances with $\Delta b > 0$.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>instances</td>
<td>1900</td>
<td>1900</td>
<td>950</td>
<td>950</td>
<td>630</td>
<td>630</td>
<td>630</td>
<td>630</td>
<td>630</td>
<td>276</td>
<td>276</td>
<td>276</td>
<td>276</td>
</tr>
<tr>
<td>nontrivial</td>
<td>682</td>
<td>819</td>
<td>575</td>
<td>751</td>
<td>130</td>
<td>42</td>
<td>244</td>
<td>152</td>
<td>238</td>
<td>187</td>
<td>214</td>
<td>267</td>
<td>271</td>
</tr>
</tbody>
</table>

Random instances. We consider point sets of size $n \in \{15, 20, 25, 30\}$. We generate ten uniformly randomly distributed sets of points for $n \in \{15, 20\}$ and five for $n \in \{25, 30\}$. For each point set, we generate 20 random triangulations (see Appendix B). Then, the flip distance problem instances corresponding to the point set are given by the $\binom{20}{2} = 190$ pairs of triangulations. Thus, for a fixed $n$ we either get 1900 ($n \leq 20$) or 950 instances ($n \geq 25$). We denote these instance sets by $RX$ in our experiments where $X$ is the size of the instance.

Sea level instances. The sea level triangulations by Nitzke et al. [28] stem from the 41 tide gauge stations in the European North Sea. The triangulations are optimized with respect to monthly generated satellite altimetry. In our experiments, we focus on the data collected from 2009 to 2011, which yields 36 triangulations in our timeframe. Nitzke et al. only consider higher-order Delaunay triangulations. Consequently, they have different sets of optimized triangulations for different orders. In the application [28], distinct orders are not mixed. Consequently, we also treat different orders as separate data sets. We focus on triangulations of order 2 and 5, which yield the most consistent reconstructions according to [28] and [1], respectively. Additionally, we consider unconstrained sea level instances in a smaller time frame of 2009–2010 (an instance is unconstrained if its order equals its size). In addition to the complete North Sea set with 41 points, we consider subsets of the tide gauge set of size $n \in \{25, 30, 35\}$, which still cover the North Sea and therefore allow for optimization. All in all, for a size $n$, we have 630 instances of order 2 and 5, respectively, and 276 unconstrained instances. We denote these instance sets by $SX-Y$ where $X$ is the size and $Y$ is the order.

Next we want to introduce a measure of hardness for an instance. For this, we define $\Delta b(D, D') = u_H(D, D') - l_E(D, D')$ to be the gap between the upper bound given by our implementation of the algorithm by Hanke et al. [13] and the best known lower bound by Eppstein. If we have $\Delta b(D, D') = 0$, the solution given by the upper bound is already optimal. We expect instances to get harder with increasing $\Delta b(D, D')$. It turns out that the sea level instances with small order are “easier” than unconstrained instances. In particular, all of the instances of sizes 25, 30 and 35 and order 2 can be trivially solved using the bounds. The numbers of non-trivial instances for the other sets are given in Table 1 and their distributions of $\Delta b(D, D')$ are given in Figure 5. Note that for order 5 all of the instances have $\Delta b(D, D') \leq 3$ and most of them have $\Delta b(D, D') = 1$. Additionally, Figure 5 suggests that the unconstrained sea level data sets contain the hardest instances.

7.2 Experimental Evaluation

Before we present our evaluation, we give some details on the used hardware and software. The systems used for the experiments are equipped with an AMD EPYC 7402P CPU, 256 GB of RAM and have Ubuntu 22.04 installed. Our code is written in C++17, compiled with GCC 9.4, and it is available online\(^2\). In our implementation, we use an array-based binary heap as priority queue. For the hashmap, we utilize the unordered map provided by the C++ standard library, while employing a simple combinatorial hash function.

---

\(^2\) https://github.com/PhilipMayer94/AStarFlipDistance
For the Hungarian algorithm, we ported the $O(n^3)$ implementation by Stern\textsuperscript{3} to C++ and added the dynamic features. As an ILP-solver Gurobi 9.5.1\textsuperscript{12} using a single thread is employed. We set the number of layers of the ILP to the flip distance $d_F$, if it is available from other experiments. Note that this information is usually not available to the ILP and it has to use an upper bound for $d_F$. Our evaluation is going to show that even this idealized version is not competitive with the $A^*$ approaches. As a baseline, we implemented an iterative bi-directional breadth-first search (BFS). We denote the $A^*$ search that uses the simple heuristic $h_S$ by $A^*_S$ and the one using Eppstein’s heuristic $h_E$ by $A^*_E$. With the clustering application in mind, we choose a timeout of five minutes in all experiments.

Comparing all approaches. We start by investigating all presented approaches. Table 2 depicts for each data set the timeout percentages of the presented algorithms (and an additional algorithm we discuss later). The BFS does not run into timeouts for the smallest data set R15 but struggles on all other sets. This can be attributed to the fact that BFS is uninformed and the runtime of BFS exponentially depends on $d_F$. The ILP performs less consistently than BFS and even has timeouts for R15, but it can at least solve some (up to 95.38\%) instances of every data set. This could be attributed to the fact that the ILP works with more information, e.g, a dual bound (which coincides with $h_S$ at the root).

Both of the $A^*$ algorithms perform more consistently than the other two approaches. They both have less than 2\% timeouts for instances with size $n \leq 20$ or order $k \leq 5$ (except for S41-5). The simpler algorithm $A^*_S$ has fewer timeouts on the easier instances with $\Delta B < 3$ and $A^*_E$ has fewer timeouts on the harder instances. Overall both approaches still have at least 25\% timeouts for the large unconstrained instances and the large random instances.

\textsuperscript{3} https://github.com/KevinStern/software-and-algorithms
Next, in Figure 6, we consider the runtimes on all solved instances (an instance is solved if at least one approach solved it in the allotted time) depending on the flip distance. Note that, if an algorithms times out for a solved instance, the timeout time is counted as its runtime. We can see that, on average, both $A^*$ algorithms outperform the other approaches by at least two orders of magnitude. On average, BFS outperforms the ILP in runtime, but as mentioned earlier, for the larger flip distances some outliers can be solved faster (or at all) with the ILP. Concerning the $A^*$ algorithms, we can see that $A^*_S$ is on average faster for $d_F < 26$ and for the larger flip distances $A^*_E$ outperforms $A^*_S$ by an order of magnitude. This is a result of the trade-off between the smaller search space of the better heuristic $h_E$ and the faster computation time for $h_S$.

We now have a brief look at the runtime-wins. Neither the ILP nor BFS are the fastest approach for a single instance. $A^*_S$ has the fastest runtime 1527 times and $A^*_E$ only 1218 times. Our previous discussion suggests that these wins depend on the hardness of the instances. We propose an additional measure $\Delta h(D_s) = h_E(D_s) - h_S(D_s)$, which quantifies how much $h_E$ improves upon $h_S$. In Figure 7 the wins are grouped with respect to $\Delta h(D_s)$. As expected most $A^*_S$ wins happen for $\Delta h(D_s) = 0$ and the rest for $\Delta h(D_s) = 1$ or $\Delta h(D_s) = 2$. When $\Delta h(D_s)$ is two or larger, $A^*_S$ consistently expands over ten times as many nodes as $A^*_E$. This observation helps to explain why $A^*_E$ exhibits faster runtimes on such instances. A more in-depth discussion of node extensions can be found in Appendix C.

**A combined approach.** Note that $\Delta h$ is apriori knowledge. Thus, we can use it to develop an approach that combines both heuristics. A first naive possibility would be to compute $\Delta h(D_s)$ and then use $A^*_S$ if $\Delta h(D_s) = 0$ and $A^*_E$ otherwise. Since this approach only covers
the $\Delta h(D_s) = 0$ wins of $A^*_S$, we engineered a more sophisticated approach $A^*_C$, in an attempt to use more $\Delta h$ knowledge. $A^*_C$ is based on the assumption that if we have $h_S(D) = h_E(D)$ for a node $D$, the heuristics will probably not diverge (too much) for children of $D$.

Thus, we make a dynamic decision at every node $D$, by considering $\Delta h(D)$, before extending it. More specifically, before line 9 in Algorithm 1 we compute $\Delta h'(D) = h_{\text{known}}(D) - h_S(D)$, where $h_{\text{known}}(D)$ is the algorithm’s best known heuristic value for $D$. If $\Delta h'(D) = 0$, we do not execute line 9 and in line 13 computeHeuristicNeighbor($D', D, D_1$) is performed with respect to $h_S$ instead of $h_E$. If $\Delta h'(D) > 0$, we perform the usual update with respect to $h_E$. This combined heuristic is valid since the maximum of two admissible and consistent heuristics still satisfies both properties. Note that performing a node extension with respect to $h_S$ implies that $h_{\text{known}}(D') = h_S(D')$ for all children $D'$ of $D$ and, if a child is later extended, it will always be extended using $h_S$.

As before, we used all of the data to test $A^*_C$. The timeouts are given in Table 2. For every data set except S41-41, the timeout percentage is smaller than that of $A^*_S$ and $A^*_E$. In particular, $A^*_C$ can solve all instances for the set S30-5 in the allotted time. The runtimes of all $A^*$ approaches grouped by flip distance are depicted in Figure 8 (a). $A^*_C$ on average computes $d_F$ faster than the other approaches for all distances. It should be noted that the runtime of $A^*_C$ is similar to $A^*_S$ for small flip distances and similar to $A^*_E$ for larger distances. This aligns with our assumption that we primarily utilize $h_S$ for easier instances and mostly rely on $h_E$ for more challenging cases. All of the previous observations can also be verified if we consider the runtime speedups of $A^*_C$ with respect to $A^*_S$ and $A^*_E$, which are given in Figure 8 (b). Here the speedup with respect to $A^*_E$ decreases with $d_F$ and the speedup with respect to $A^*_S$ increases. We can also deduce from the speedup plot that dynamically choosing the heuristic in $A^*_C$ improves upon $A^*_E$ for hard instances since otherwise, we would expect the speedup for $A^*_E$ to be very close to one for $d_F>30$.

Next, we consider the runtime-wins grouped by $\Delta h(D_s)$, which are given in Figure 9. Note that if two algorithms have the same (fastest) runtime on an instance, the win is counted for both of them. Algorithm $A^*_C$ has by far the most wins, except for $\Delta h(D_s) = 0$, but on
those instances $A^*_C$ mimics $A^*_S$. Thus, all of those wins are actually “shared” wins where both approaches have the same runtime. In fact, $A^*_S$ only solves 1.1% and $A^*_E$ only 6.5% of the instances faster than $A^*_C$. Hence, for 92.4% of instances $A^*_C$ is the (possibly shared) fastest algorithm and for 43.8% instances it even improves upon the basic $A^*$ algorithms. On average our algorithm $A^*_C$ is 20% faster than the virtual best of $A^*_S$ and $A^*_E$ (the virtual best uses a posteriori knowledge to choose the faster of the two approaches for every instance).

Finally, we discuss the peak memory consumption of the algorithm. On the most memory-intensive instance, which consists of 41 points and times out after five minutes with 201 109 543 opened triangulations, the combined algorithm uses 50 GB of memory. Most of this is due to the hashmap that manages the open triangulations. Its memory usage could be improved by changing the representation of the triangulations to a more sophisticated one, such as an adaptation of the planar encodings given by Chuang et al.\cite{5}. Note that maintaining these encodings may come with an increase in runtime. Additionally, the unordered map of the C++ Standard Library has a large memory overhead. Thus, using a more memory-efficient map could also improve memory usage.

A brief evaluation of the decomposition heuristic. We ran our decomposition-based heuristic on all of the application-relevant instances, i.e., the sea level instances with order 2 and 5. On all instances that could be solved by $A^*_C$ in the allotted time, our heuristic value coincided with the optimal flip distance value computed by $A^*_C$. The average runtime of the heuristic on the considered instances was 0.66 milliseconds. This suggests that, at least for the considered application-oriented instances, finding an optimal solution is usually not difficult, but verifying its optimality takes significant effort. Finally, to get an idea of how the runtime increases for larger instances, we generated 100 random instances of order 5 for $n = 50, 100, \ldots, 500$ and computed our heuristic on them. The runtimes are given in Figure 10. The increase in runtime on the considered data set is close to linear and even instances of size $n = 500$ can on average be solved heuristically in 30 milliseconds.

8 Conclusion

We have engineered an $A^*$ algorithm that combines two different heuristics and discussed how to dynamically compute them. Our approach significantly outperforms bidirectional breadth-first search as the baseline, as well as our novel integer linear program. Random instances of size up to 25 points can consistently be solved by our algorithm. Our investigation of the instances from the sea surface reconstruction task shows that our approach can solve over 90% of the relevant instances (41 points in the North Sea) with small Delaunay orders in less than five minutes per instance.
On all of our considered application data, our new decomposition-based heuristic rapidly computes a solution that is already optimal, suggesting that in our case (sea surface reconstruction under higher-order Delaunay constraints) possibly only verifying optimality is hard. Overall, with our new optimal algorithm the clustering task in the context of sea surface reconstruction may become feasible for small data sets (e.g., the North Sea), and using our heuristic approach it would also be feasible for the global data sets (with more than 500 points [1]). Employing them here might yield interesting results.

References


A Eppstein’s Heuristic is Admissible and Consistent

Proposition 3. The heuristic \(h_E(D) = l_E(D, D_t)\) is admissible and consistent.

Proof. We start by showing that the heuristic is admissible. This has already been shown by Eppstein [9], but we reiterate the proof for the sake of completeness. By Observation 2 the length of a minimal flip sequence between \(D\) and \(D_t\) is given by the sum of lengths of the walks in \(QG\). The length of every walk is lower bounded by the length of the shortest path between its endpoints. The shortest paths correspond to a perfect matching in \(B(D, D_t)\) and thus, the sum of their lengths is lower bounded by the value of the MWPM in \(B(D, D_t)\) which is exactly \(h_E(D) = l_E(D, D_t)\); see Figure 2 for an example.

It remains to be shown that \(h_E\) is consistent. If all diagonals coincide, there is a perfect matching of length zero which implies \(h_E(D_t) = l_E(D_t, D_t) = 0\).

Let \(D\) and \(D'\) be adjacent triangulations in \(FG(S)\). Since they are adjacent they differ by exactly one flip of the diagonal \(e\). Thus, the diagonal \(e\) was moved in the \(QG\) graph to an adjacent diagonal \(e'\). Consequently, in length, the shortest paths from \(e'\) to the diagonals in \(D_t\) can differ to the paths from \(e\) by at most one.

If we now assume that \(e\) and \(e'\) correspond to the same node \(v\) in \(B(D, D_t)\) and we define \(c\) to be the costs with respect to \(D\) and \(c'\) to be the costs with respect to \(D'\). Then, for all \(x' \in D_t\) we have \(c(x, x') = c'(x, x')\) for all \(x \neq v\) and \(|c(x, x') - c'(x, x')| \leq 1\) for \(x = v\). Now let \(M'\) be the MWPM for \(c'\) and \(M\) for \(c\). We get

\[c'(M') \geq c(M') - 1 \geq c(M) - 1,\]

which implies \(h_E(D) \leq h_E(D') + 1\).

B (Our) Random Triangulations for Point Sets

In this section we briefly discuss how to generate random triangulations. To the best of our knowledge, there is no “easy” way to generate triangulations of point sets uniformly at random. We use the following approach:

1. Generate a set \(S\) of points uniformly at random in \([0, 1]^2\).
2. For every triangle \(T\) that can be used in some triangulation of \(S\), generate a weight \(w_T\) uniformly at random in \([0, 1]\).
3. Compute the triangulation \(D_{\text{rand}}\) that minimizes the sum of its triangle weights, i.e.,

\[D_{\text{rand}} = \arg\min_{D \in \mathcal{T}(S)} \sum_{T \in D} w_T.\]  \hspace{1cm} (1)

4. Repeat Steps 2 and 3 if multiple random triangulations for the same point set are needed.

Note that Step 3 is NP-hard, but for instances with up to 500 points the triangulation can usually be found quickly using an integer linear program. We want to emphasize, that this approach not necessarily generates triangulations picked uniformly at random, but for our use case it yields an interesting mix of easy and hard instances.
C A Brief Look at Node Extensions

We also considered the number of node extensions during our experiments. Again, we used all non-trivial instances of the datasets. In Figure 11 we present an overview of the node extensions grouped by the flip distance. In (a) the total numbers of node extensions are given and in (b) the ratios of extended nodes $A^*_S/A^*_C$ and $A^*_E/A^*_C$ are depicted. Notice that the numbers of extended nodes for $A^*_S$ are the largest which immediately follows from the fact that Eppstein’s heuristic is at least as accurate as the simple heuristic. Starting at flip distance 30 the number of extensions with $A^*_S$ is five to ten times larger than the number of extensions for $A^*_C$ and $A^*_E$. This can also be seen in the ratio-plot (b).

Up to this point, our discussion suggests that with increasing flip distance the node extension ratio increases. This is of course somewhat true, but there is a second aspect namely the quality difference of the heuristics that needs to be considered. We focus on the heuristic distance $\Delta h$ at the root/start triangulation $D_s$. Most instances with large flip distances also have large $\Delta h$. A visualization of the ratio of extended nodes $A^*_S/A^*_E$ grouped by $\Delta h$ is given in Figure 12. As expected, we see that for $\Delta h = 0$ the ratio is close to one. For $\Delta h = 1$ it is between two and five and starting at $\Delta h = 2$, it is larger than ten.

This explains the runtime-wins that $A^*_S$ achieves for $\Delta h < 2$, since for those instances the decreased number of node extensions for $A^*_E$ with respect to $A^*_S$ is not enough to compensate for the increased runtime of the heuristic calculation of $h_E$.

![Figure 11](image-url) The number of node extensions of the $A^*$ approaches on all considered random and real-world data. Only solvable and non-trivial instances were considered. In (a), the total numbers of node extensions are given, and in (b) the node extension ratios of $A^*_S$ and $A^*_E$ with respect to $A^*_C$ are given grouped by flip distance (uneven $d_F$ values are grouped with their even predecessors).
Figure 12 The ratio of extended nodes of $A^*_S$ with respect to $A^*_E$, grouped by the heuristic distance $\Delta h$ at the root/start triangulation $D_s$. 
Efficient Exact Online String Matching Through Linked Weak Factors

Matthew N. Palmer
The British Computer Society, Swindon, United Kingdom

Simone Faro
Department of Mathematics and Computer Science, University of Catania, Italy

Stefano Scafiti
Department of Mathematics and Computer Science, University of Catania, Italy

Abstract

Online exact string matching is a fundamental computational problem in computer science, involving the sequential search for a pattern within a large text without prior access to the entire text. Its significance is underscored by its diverse applications in data compression, data mining, text editing, and bioinformatics, just to cite a few, where efficient substring matching is crucial. While the problem has been a subject of study for years, recent decades have witnessed a heightened focus on experimental solutions, employing various techniques to achieve superior performance. Notably, approaches centered around weak factor recognition have emerged as leaders in experimental settings, gaining increasing attention. This paper introduces Hash Chain, a novel algorithm founded on a robust weak factor recognition approach that links adjacent factors through hashing. Building upon the efficacy of weak recognition techniques, the proposed algorithm incorporates innovative strategies for organizing data structures and optimizations to enhance performance. Despite its quadratic worst-case time complexity, the new proposed algorithm demonstrates sublinear behavior in practice, outperforming currently known algorithms in the literature.

2012 ACM Subject Classification Theory of computation → Bloom filters and hashing; Theory of computation → Pattern matching

Keywords and phrases String matching, text processing, weak recognition, hashing, experimental algorithms, design and analysis of algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.24

Funding This work is funded by the National Centre for HPC, Big Data and Quantum Computing, Project CN00000013, affiliated to Spoke 10.

1 Introduction

The string matching problem consists in finding all the occurrences of a pattern $x$ of length $m$ in a text $y$ of length $n$, both strings defined over an alphabet $\Sigma$ of size $\sigma$. The relevance of this problem is due to its direct applications in many fields, such as data compression, data mining, text editing, and bioinformatics, where fast and efficient pattern matching is critical. Given its fundamental nature in computer science, marked by intriguing theoretical and practical aspects that make it particularly compelling and challenging, the string matching problem has garnered significant attention in the literature.

The problem can be addressed in online mode, when you do not have access to the text before carrying out the search, or in offline mode, in which case it is possible to preprocess the text to speed up the search phase. In this work we consider online string matching.

1 Corresponding author
Online string matching can be solved in linear time, with respect to the size of the text, but although the first optimal solutions appeared already in the 1970s, such as the Knuth, Morris and Pratt [22] (KMP) algorithm, several solutions have been proposed over the past decades [15], many of which appeared in recent years [18, 19]. This suggests that the interest in increasingly efficient solutions has been significant since the beginning and is still high.

Boyer and Moore (BM) provided the first sub-linear solution on average [4], while the Backward-Dawg-Matching (BDM) algorithm [9] was instead the first solution to reach the optimal $O(n \log_\sigma (m)/m)$ time complexity on the average, as proved by Yao [27]. Both the KMP and the BDM algorithms are based on an exact factor recognition: they simulate the deterministic automaton for the language $\Sigma^*$ and the deterministic suffix automaton for the language of the suffixes of $x$, respectively. Many of the papers in the literature have amply demonstrated (see for instance [25, 10, 6, 14]) how the efficiency of these solutions is strongly affected by the encoding used for simulating the underlying automaton, especially when the bit-parallelism technique [2] is used. For this reason, recent research [11, 18, 7, 10] has focused more on approaches based on a weak recognition.

An algorithm performs weak recognition when it is able to recognize a broader language than the set of pattern sub-strings. The Backward Oracle Matching [1] (BOM) can be considered the pioneer of this approach, making use of the Factor Oracle of the reverse pattern. In the same direction, the Weak Factor Recognition algorithm [7] (WFR) approach is based on indexing all the $O(m^2)$ subsequences of the pattern $x$ using a bloom filter [3]. In [10], Q-gram Filtering (QF) ensures that $q$-grams read in a window all belong to the same chain of $q$-grams in the pattern. More recently, Faro and Scafiti introduced the Range Automaton [18], a non-standard, weak version of the non-deterministic suffix automaton.

In this paper, we introduce the Hash Chain algorithm (HC), a new efficient algorithm for online exact string matching based on weak factor recognition and hashing. The HC algorithm stands out for its refined experimental design, marked by distinctive features and optimizations that elevate its practical performance to exceptional levels. It is based on a robust improved filtering approach which links together hash values corresponding to adjacent factors of the input string $x$. Despite the $O(nm)$ worst-case time complexity, the HC algorithm exhibits a sublinear behaviour in practice, obtaining the best running times when compared against the most effective algorithms known to date in the literature. This makes our proposed solution one of the most flexible algorithms in practical cases.

The paper is organized as follows. In Section 2 we introduce the new algorithm, describing its preprocessing and searching phase in detail. In Section 3 we propose some practical optimizations to improve the performance of the algorithm. Finally, we present in Section 4 the results of an extensive experimentation. We draw our conclusions in Section 5.

## 2 The Hash Chain Algorithm

In this section we present an efficient algorithm for the exact string matching problem based on a weak-factor-recognition approach using hashing. Our proposed algorithm is named Hash Chain (HC) and consists of a preprocessing and a searching phase. The algorithm is influenced by the Weak Factor Recognition (WFR) algorithm [8], which employs indexing via a Bloom filter to store hash values derived from all pattern factors. This implies that if, during the search phase, two consecutive text factors are identified that match portions of the pattern but are not in adjacent positions, the search continues even if an exact match will not be found. The HC algorithm also incorporates information about adjacent factors into its data structure to streamline the verification phase and drastically decrease the occurrence of false positives. In addition, by identifying $q$-grams which are not adjacent to each other in the pattern, the HC algorithm enables a large forward shift on average.
Before delving into the details, it is essential to formalize certain concepts and introduce key definitions that will prove valuable throughout the paper. We represent a string \( x \in \Sigma^m \) as an array \( x[0...m-1] \) of characters of \( \Sigma \) and write \( |x| = m \). For \( m = 0 \) we obtain the empty string \( \varepsilon \). Thus, \( x[i] \) is the \((i+1)\)-st character of \( x \), for \( 0 \leq i < m \), and \( x[i..j] \) is the substring of \( x \) contained between its \((i+1)\)-st and the \((j+1)\)-st characters, for \( 0 \leq i \leq j < m \).

A \( q \)-gram of \( x \) is a substring of \( x \) with a fixed length \( q \). Given a finite alphabet \( \Sigma \), we denote by \( \Sigma^m \), with \( m \geq 0 \), the set of all strings of length \( m \) over \( \Sigma \) and put \( \Sigma^* = \bigcup_{m \in \mathbb{N}} \Sigma^m \).

Throughout the paper we assume \( x \) is a pattern of length \( m \) and \( y \) is a text of length \( n \). We also assume that both strings \( x \) and \( y \) are drawn from a common alphabet \( \Sigma \) of size \( \sigma \). In addition, we make use of the following bitwise operators: the bitwise OR \( \lor \), the bitwise AND \( \land \), and bit-shift left \( \ll \).

We are now ready to provide a detailed description of the Hash-Chain algorithm, discussing, in the following three sections, the preprocessing phase, the structure of the hash function, and the searching phase, respectively.

### 2.1 The preprocessing phase

The preprocessing phase consists of the computation of an extended Bloom filter data structure indexing all the \( q \)-grams of the pattern \( x \). It is backed by a bit-vector \( F \) of \( 2^\alpha \) words, where each word has \( w \) bits and \( \alpha \) controls the size of \( F \). Two hash functions are used: \( h : \Sigma^q \rightarrow \{0,1,...,2^\alpha - 1\} \), which produces an index into a word in \( F \), and \( \lambda : \{0,1,...,2^u - 1\} \rightarrow \{0,1,...,2^w - 1\} \), which outputs a word with only one of its bits set.

The filter is built by linking together each pair of adjacent non-overlapping factors, \( u_1 \cdot u_2 \) of fixed size \( q \) using the following formula:

\[
F[h(u_2)] \leftarrow F[h(u_2)] | \lambda(h(u_1)).
\]

(1)

We use the bitwise OR operator \( \lor \) to retain bits already set from previous \( q \)-gram pairs, if they had hashed to the same word in \( F \). More formally, formula 1 is iterated for each pair of distinct \( q \)-grams \( \langle u_1, u_2 \rangle \) such that:

- \(|u_i| = q, i = 1, 2, \)
- \(u_1 = x[i..j], \) where \( j = i + q - 1, \) for some \( 0 \leq i \leq m - 2 \cdot q \)
- \(u_2 = x[j+1..j+q]\)

Note that the first \( q \) positions of any pattern do not have a \( q \)-gram to their left. To ensure they are recognized as factors of the pattern, we index any such factor \( u \) separately using the following formula:

\[
F[h(u)] \leftarrow F[h(u)] | 1\]

(2)

To apply formula 1 efficiently to each pair of non-overlapping adjacent factors of \( x \), Hash Chain groups sequences of non-overlapping \( q \)-grams according to the position they appear in the pattern. More specifically, each pattern position \( j \), with \( m - q \leq j < m \), defines a sequence of \( \lfloor (j + 1)/q \rfloor \) non-overlapping \( q \)-grams, given by:

\( \{x[i..i + q - 1] | i \geq 0, i = j - q + 1, j - 2q + 1, ... \} \).

Each of such sets is denoted as a \( q \)-gram chain of the pattern. By processing \( q \)-grams in chains, we can pass a hash value from one linked pair of \( q \)-grams to the computation of the next pair. This approach enables us to compute the hash only once for each \( q \)-gram in the pattern. Figure 1 shows the 3 chains of \( q \)-grams arising in a pattern of length \( m = 13 \) with \( q = 3 \).
Efficient Exact Online String Matching Through Linked Weak Factors

Figure 1 The three q-gram chains for a pattern x of length \( m = 13 \) and \( q = 3 \), and the operations performed for each q-gram. The \( |= \) symbol represents the logical Or operator (the result is stored in the left hand operand). Each hash \( H_n \) is labelled with a number \( n \) reflecting the order in which they are calculated during pre-processing.

Obviously, a pattern where \( m = q \) can only have one chain, and one q-gram, in it. More generally, when \( m < 2 \cdot q - 1 \), it only has \( m - q + 1 \) distinct chains, and when \( m \geq 2 \cdot q - 1 \), it has \( q \) distinct chains of q-grams.

Figure 2 shows the process of linking four adjacent q-grams of the pattern \( x = \text{acgtagcgtgcaca} \). To complete pre-processing, we would need to process the remaining chains of q-grams in pattern \( x \): those starting at position 1 (\( \text{cgtg}, \text{tacg}, \text{ctgc} \)), the ones at position 2 (\( \text{gtgt}, \text{acgc}, \text{tgca} \)) and those at position 3 (\( \text{tgta}, \text{cgct}, \text{gcac} \)).

2.2 Hash functions

As described above, the preprocessing phase makes use of two different hash functions, and specifically the HASH function \( h : \Sigma^q \rightarrow \{0, 1, \ldots, 2^\alpha - 1\} \), which produces an index into a word in \( F \), and the LINK-HASH function \( \lambda : \{0, 1, \ldots, 2^\alpha - 1\} \rightarrow \{0, 1, \ldots, 2^w - 1\} \), which outputs a word with only one of its bits set. The design of \( h \) and \( \lambda \) strongly affect the performance of the algorithm and, for this reason, has been carefully defined.

The HASH function \( h \) has been designed to follow a shift-then-add structure, which can be implemented efficiently in modern architectures [26]. It is recursively defined as:

\[
h(x) = \begin{cases} 
0 & \text{if } m = 0 \\
(h(x[1..m-1]) : 2^s + x[0]) \mod 2^\alpha & \text{otherwise.}
\end{cases}
\] (3)

The hash value is multiplied by \( 2^s \), or equivalently bit-shifted left by \( s \), for each additional character added. To ensure we do not shift data much beyond the maximum hash value, we calculate the bit shift \( s \) by the following formula:

\[ s \leftarrow [\alpha/q] \] (4)
The final value of the hash is taken as \( \text{mod } 2^\alpha \), which can be efficiently computed by bitwise ANDing it with \( 2^\alpha - 1 \).

The \text{LINK-HASH} function \( \lambda \) is a simple function mapping each value \( 0 \leq v < 2^w \) to the set \( \{2^0, 2^1, \ldots, 2^{w-1}\} \). It is meant to link together adjacent factors of the input pattern \( x \), and that’s why we refer to it as the link hash function. Its definition is given by:

\[
\lambda(v) = 2^{(v \mod w)},
\]

where \( v \) is the value to obtain a link hash for and \( w \) is the number of bits in a word in the bit vector \( F \). Given that \( w \) is a power of two, the \( \text{mod } w \) operation can be efficiently computed by logically bitwise ANDing it with \( w - 1 \). It returns a word with a single bit set in it.

Pseudo-codes for the hash function \( h \) (\text{HASH}), the link hash function \( \lambda \) (\text{LINK-HASH}), and the \text{PREPROCESSING} procedure are given in Figure 3. We don’t pass \( w \) into the \text{LINK-HASH} function as it is assumed to be hard-coded. Note that the \text{PREPROCESSING} procedure processes each chain of \( q \)-grams backwards, calculates the hashes for the first \( q \) \( q \)-grams last, and returns a hash value \( H_v \) in order to facilitate some optimisations discussed in section 3. It uses \text{min}() functions on lines 5 and 14 to ensure that we only process \( q \)-gram chains that actually exist in the pattern, as short patterns where \( m < 2 \cdot q - 1 \) have fewer than \( q \) chains.

Obviously, a pattern where \( m = q \) can only have one chain, and one \( q \)-gram, in it; more generally when \( m < 2 \cdot q \), it has \( m - q + 1 \) distinct chains of a single \( q \)-gram each.

Regarding the time complexity of the preprocessing phase, it is proportional to the number of \( q \)-grams in the pattern. A pattern contains \( m - q + 1 \) distinct \( q \)-grams in it, each of which requires a hash computing for it once if the \( q \)-grams are processed in chains. It also re-computes the hash for the first \( q \) \( q \)-grams again to set a bit for them. Each \( q \)-gram requires \( O(q) \) time to be read, so the time complexity is \( O(m \cdot q) \).
Figure 3 (On the left) The pseudo-codes of the Hash function \( h : \Sigma^q \rightarrow \{0, 1, \ldots, 2^\alpha - 1\} \), which produces an index into a word in \( F \), and the Link-Hash function \( \lambda : \{0, 1, \ldots, 2^w - 1\} \rightarrow \{0, 1, \ldots, 2^w - 1\} \), which outputs a word with only one of its bits set. (On the right) The pseudo-code of the preprocessing phase of the Hash-Chain algorithm.

Figure 4 The general approach of factor based search algorithms. A factor \( u \) is read backwards from the end of the current window until \( u_p u \) is not a factor of the pattern; it is then safe to shift the window past \( u_p \). If the entire window is read, then a possible match must be verified when a weak recognition approach is used.

2.3 The Searching Phase

The searching phase works like any factor algorithm, such as BOM, WFR or QF. The difference between them lies in how valid factors of the pattern are determined.

A window of size \( m \) is slid along the text, starting at position 0, and shifted to the right after each attempt, until we reach the end of the text. A factor \( u \) of the pattern is read backwards in the text \( y \), from the position aligned with the end of the window. If \( u_p u \) is not a factor of the pattern, then it is safe to shift the window after \( u_p \). This is shown in figure 4.
The pseudocode of the HC search algorithm is given in figure 5. We determine whether a \( q \)-gram is a valid factor \( u \) by first calculating its hash value \( v \). For the first \( q \)-gram processed, aligned with the end of the current window, it looks in the word at \( F[v] \) to see if it is empty at line 6. If a word in \( F \) is empty, no \( q \)-gram in the pattern hashed to \( v \), and so \( u \) cannot be a factor. We then shift on from the end of the window by the maximum distance it is possible to shift without missing a possible occurrence, \( m - q + 1 \), and look at the next window.

If \( F[v] \neq 0 \), then we have a possible match for \( u \) and the algorithm enters the while else loop at line 8. Hash chain must now look in turn at the \( q \)-grams in the window preceding \( u \) to see if they are also possible factors of the pattern. Since pre-processing links adjacent \( q \)-grams together with the \( \lambda \) function, we calculate the hash value \( v_{p} \) of the preceding factor \( u_{p} \), and check to see if the bit returned by \( \lambda(v_{p}) \) is set in \( F[v] \) at line 11. If the bit is not set, then the two factors were not linked during pre-processing, and \( u_{p}u \) cannot be a factor of the pattern. It is then safe to shift from the position of \( u_{p} \) and look at the next window.

This is repeated until we reach the start of the current window, when the else branch at line 14 executes. On line 15, the position of the search \( j \) is updated such that adding \( j + m - q + 1 \) to it on line 18 results in the window being advanced only by one. Finally, a check for the existence of the pattern is executed from line 16 to verify the actual presence of the pattern before reporting any occurrence. Note that, according to the while else semantics., the else branch at line 14 is only taken if the execution of the loop has not been interrupted by the break statement at line 12: naive check is thus not executed unless the window has been fully scanned.

The window is eventually advanced at line 18 depending on the size of the scanned window, thus starting a new iteration of the algorithm.

Concerning the algorithm’s complexity, it is evident that the search phase exhibits a time complexity of \( O(nm) \) in the worst-case scenario and necessitates additional space of \( O(2^{\alpha}) \). With the preprocessing phase consuming computational time at \( O(mq) \), the overall complexity of the algorithm is delineated as \( O(m(n + q)) \). While its complexity may not rival other solutions that demonstrate superior worst-case efficiency, the proposed algorithm manifests commendable performance in practical scenarios, as underscored by our experimental tests elucidated in Section 4.

### 3 Algorithmic Optimisations

Optimizing algorithms is a crucial aspect in algorithm design, particularly in experimental contexts. The efficiency of an algorithm not only impacts its practical applicability but also influences resource consumption, execution speed, and overall performance. In experimental settings, where algorithms are tested and evaluated, optimization becomes paramount to ensure that computational resources are utilized effectively.

In this section we discuss the following optimisations for the Hash Chain algorithm, which are implemented in the bench-marked versions: (i) reduce the bits set for the first \( q \) \( q \)-grams; (ii) reduce need for full pattern verification; (iii) use a sentinel pattern at the end of the text.

The first two optimisations are included in the basic Hash Chain (HC) algorithm, while the Sentinel Hash Chain algorithm (SHC) additionally implements the last one.

#### 3.1 Reduce the Bits Set for the First \( q \) \( q \)-grams

The preprocessing order originally presented in Figure 2 is sub-optimal in one way. The first step was to place an arbitrary 1 bit into the hash address of the first \( q \)-gram, to ensure it is recognised as a factor. This simplified the description process; we start at the beginning and progressed straightforwardly to the end.
Efficient Exact Online String Matching Through Linked Weak Factors

Hash-Chain$(x, m, y, n, q, \alpha)$
1. $F, s, F', H_v \leftarrow \text{Preprocessing}(x, m, q, \alpha)$
2. $j \leftarrow m - 1$
3. while $j < n$ do
4. $v \leftarrow \text{Hash}(y, j, q, s, F')$
5. $z \leftarrow F[v]$
6. if $z \neq 0$ then
7. $i \leftarrow j - m + 2 \cdot q$
8. while $j \geq i$ do
9. $j \leftarrow j - q$
   if $z \& \text{LinkHash}(v) = 0$ then
10. break (to line 18)
11. else
12. $j \leftarrow i - q$
13. if $v = H_v$ and $y[j..j - q + m - 1] = x$
14. output $j - q$
15. else
16. $j \leftarrow j + m - q + 1$

Figure 5 The pseudo-code of the Hash-Chain (HC) algorithms for the exact string matching problem. In addition to the two strings and their length, the algorithm receives as input the size of the $q$-grams used for hashing and the parameter $\alpha$ which regulates the size of the hash table.

However, the algorithm only requires that the word in $F$ for that $q$-gram is not empty, in order that it can be identified as a valid factor of the pattern. If we process all the other pairs of $q$-grams first, it is possible that a collision will occur and the entry for it will already contain one or more bits. In that case, there is no need to set an additional 1 into the entry, as it already flags that it is a possible factor, by not being empty.

Therefore, when implementing the pre-processing phase, it is advisable to process the first $q$ $q$-grams with no preceding $q$-gram last, and to only place a bit into the entry for them if it is empty. This strategy ensures we set as few bits as possible in the bit vector, which reduces the chance of a false positive match. The pre-processing pseudo-code given in Figure 3 already implements this optimisation in lines 14-17.

3.2 Reduce the Full Pattern Verifications

When the algorithm reads back to the start of the current window, it always performs a full pattern verification. However, note that if the pattern does match the current window, then the last hash value $v$ calculated on line 10 of the pseudocode in Figure 5 must match the hash value of the first $q$-gram in the chain ending at the end of the pattern, which we will call $H_v$. Therefore, if we return $H_v$ from the pattern pre-processing stage, we can compare $v$ with it before entering the pattern verification step at line 16. If the hash does not match, there is no need to perform full pattern verification.

This is the reason for the ordering of pre-processing in the pseudo-code in Figure 3. Each chain of the pattern is processed backwards from the end of the pattern. The last chain to be processed ends at the end of the pattern. We compute that chain back from the end of the pattern, so the last hash value computed in it is the first hash in that chain, which is $H_v$. This can then be returned by the pre-processing stage without having to re-compute it. The pseudo code for HC search in figure 5 shows this optimisation on line 16, where we test that $v = H_v$ before attempting to verify that the text matches the pattern. In Figure 1, the hash value $H_{11}$ as the first $q$-gram in the first chain would be returned as $H_v$. 
Figure 6 The pseudo-code of the Sentinel-Hash-Chain (SHC) algorithms for the exact string matching problem. In addition to the two strings and their length, the algorithm receives as input the size of the \( q \)-grams used for hashing and the parameter \( \alpha \) which regulates the size of the table.

3.3 Use a Sentinel Pattern at the End of the Text

A final optimisation technique, that can be applied to many different algorithms, is the use of a sentinel pattern at the end of the text. This technique first makes a copy of the pattern into the text, just after the end of the text to be searched, called the **sentinel pattern**. When searching, it uses a fast search loop that does not have to perform a position check. This is because the sentinel pattern at the end of the text guarantees we will find a copy of the pattern if we go past the end of the text, so we can safely loop without checking our position. Once the fast loop exits, we have to check that we have not run past the end of the text, but if not, we have a possible match to consider.

This technique, while powerful, has some serious constraints for real-world use. It requires control over the memory allocation of the text buffer to be searched, and the ability to write data into it. Many real-world applications will not offer that control to a search algorithm, but in cases where it is possible, it can have a performance advantage. It has been implemented and bench-marked separately as the Sentinel Hash Chain algorithm (SHC).

Pseudo code for the SHC algorithm is given in figure 6. The pattern \( x \) is copied to the end of \( y \) at \( n \) in lines 2 and 3. The fast loop without a position test looking for blank words in \( F \) is at lines 6 and 7, and we test to see if we have run past the end of the text at \( n \) in line 8. If not, we proceed to validate the rest of the chain and the pattern as normal.

4 Experimental Results

We report in this section the results of an extensive experimental comparison of the HC algorithm against the most efficient solutions known in the literature for the online exact string matching problem. Specifically, the following 21 algorithms (implemented in 99 variants, depending on the values of their parameters) have been compared:
24:10 Efficient Exact Online String Matching Through Linked Weak Factors

- AOSO<sub>q</sub>: Average-Optimal variant [21] of Shift-Or [2] with \(2 \leq q \leq 6\);
- BNDM<sub>q</sub>: Backward-Nondeterministic-DAWG-Matching [24] with \(1 \leq q \leq 6\);
- BRAM<sub>q</sub>: Backwards Range Automaton [18], with \(3 \leq q \leq 7\);
- BSDM<sub>q</sub>: Backward-SNR-DAWG-Matching [14], with \(2 \leq q \leq 8\);
- BSX<sub>q</sub>: Backward-Nondeterministic-DAWG [10], with \(1 \leq q \leq 8\);
- FJS algorithm [20];
- LBNDM: Long BNDM algorithm [25];
- KBNDM: Factorized BNDM algorithm [6];
- FSB<sub>q,s</sub>: Forward Simplified [6] BNDM [24], with \(2 \leq q \leq 8\) and \(1 \leq s \leq 6\);
- HASH<sub>q</sub>: Hashing algorithm [23], with \(3 \leq q \leq 8\);
- HC<sub>q,α</sub>: Hash Chain, and its variant SHC<sub>q,α</sub>, with \(1 \leq q \leq 8\) and \(8 \leq α \leq 12\);
- IOM and WOM: Improved Occurrence and Worst Occurrence Matching [5];
- QF<sub>q,s</sub>: Qgram-Filtering algorithm [10], with \(2 \leq q \leq 16\) and \(1 \leq s \leq 6\);
- SBNDM<sub>q</sub>: Simplified BNDM [28] with \(2 \leq q \leq 8\);
- WFR<sub>q</sub>: Weak Factor Recognition [8], with \(1 \leq q \leq 8\) and its variant TWFR<sub>q</sub>;
- UFM<sub>q</sub>: Unique Factor Matcher [17], with \(1 \leq q \leq 10\).

For completeness, we also included the Exact Packed String Matching (EPSM) algorithm [12], which makes use of SSE or AVX instructions and which can only report counts but not the positions of occurrences. Although we report its timings, we do not compare it with the other algorithms.

All algorithms have been implemented in the C programming language and have been tested using the Smart tool [16]. All experiments have been executed locally on a computer running Linux Ubuntu 22.04.1 with an Intel Xeon E3-1226 v3 CPU @ 3.30GHz and 24GB ECC RAM.

Our tests have been run on a genome sequence, a protein sequence, and an English text (each of size 100MB) extracted from the well known Pizza & Chilli Corpus<sup>3</sup>. In the experimental evaluation, patterns of length \(m\) were randomly extracted from the sequences, with \(m\) ranging over the set of values \(\{2^i \mid 3 \leq i \leq 9\}\). In all cases, the mean over the search speed plus the pre-processing time (expressed in milliseconds) of 500 runs for each pattern length has been reported.

Tables 1, 2 and 3 summarise our evaluations. Each table is divided into five blocks. The first block contains algorithms based on automata. The second contains algorithms based on character comparison. The third block contains algorithms which use weak factor recognition, which includes the Hash Chain algorithm. The fourth block contains algorithms that modify the text buffer to use a “sentinel” optimisation technique; all of these are also weak factor algorithms. The final block contains algorithms which are limited to only reporting a count of occurrences, but not their positions. Results within 105% of the best time are underlined, and best results have been boldfaced (without considering EPSM in the final block). For algorithms with variant parameters, such as the \(q\)-gram length, only the fastest variant is presented in brackets in a subscript next to the result.

Time measurements are denoted in milliseconds (ms). Nonetheless, in certain instances, certain solutions exhibited notable delays. Consequently, entries marked with the symbol “\(> 200\)” signify that the algorithm exceeded 200ms in search time and experienced a timeout.

---

<sup>2</sup> The source code for the new algorithm and the Smart tool are available for download respectively at https://github.com/nishihatapalmer/HashChain and https://github.com/smart-tool/smart.

<sup>3</sup> The corpus is available at http://pizzachili.dcc.uchile.cl/index.html.
M. N. Palmer, S. Faro, and S. Scafiti

24:11

Table 1 Experimental results obtained for searching on a genome sequence. Entries that state
> 200 indicate that the algorithm took longer than 200ms to search and timed out.
m

8

16

32

64

128

256

512

AOSOq
BNDMq
BSDMq
BXSq
EBOM
FSBq,s
KBNDM
LBNDM
SBNDMq

63.17(2)
37.48(4)
29.0(4)
37.06(4)
95.02
35.1(4,1)
107.38
140.1
36.8(4)

38.65(4)
19.63(4)
15.59(6)
19.04(4)
64.79
19.13(6,2)
68.98
80.12
19.03(4)

19.66(6)
10.35(6)
9.3(7)
9.72(6)
40.09
10.4(6,1)
39.31
45.44
10.5(6)

19.75(6)
10.24(6)
7.45(8)
9.74(6)
24.51
10.38(6,1)
23.83
36.13
10.52(6)

19.71(6)
10.28(6)
7.19(8)
9.71(6)
14.58
10.47(6,1)
23.2
30.05
10.49(6)

19.74(6)
10.26(6)
7.27(7)
9.73(6)
9.63
10.49(6,1)
23.22
>200
10.49(6)

19.66(6)
10.19(6)
7.34(7)
9.71(6)
7.75
10.35(6,1)
23.19
>200
10.4(6)

FJS
HASHq
IOM
WOM

>200
88.0(3)
>200
>200

>200
42.34(3)
>200
>200

>200
21.85(5)
>200
>200

>200
12.86(5)
>200
97.49

>200
11.12(5)
>200
84.66

>200
11.68(5)
>200
72.05

>200
13.73(5)
>200
64.32

BRAMq
HCq,α
QFq,s
UFMq
WFRq

58.56(5)
30.3(4,12)
33.4(4,3)
42.0(5)
35.6(4)

25.13(5)
14.2(6,12)
14.69(4,3)
18.38(6)
16.63(5)

12.67(7)
8.5(6,12)
8.66(6,2)
9.96(7)
9.93(5)

8.76(7)
7.14(6,12)
7.39(6,2)
7.71(8)
7.72(7)

7.83(7)
7.06(6,12)
7.08(6,2)
7.68(8)
7.17(6)

6.92(7)
5.75(6,12)
5.77(6,2)
6.46(8)
5.79(6)

4.03(7)
3.35(6,12)
3.43(6,2)
3.66(8)
3.25(7)

SHCq,α
TWFRq

29.6(4,12)
31.04(4)

12.8(5,12)
15.68(5)

8.7(6,12)
9.33(6)

7.39(6,12)
7.62(6)

7.03(6,12)
7.08(6)

5.63(6,12)
5.66(6)

3.29(6,12)
3.17(7)

EPSM

22.93

9.96

6.82

6.63

5.57

3.5

1.98

Table 2 Experimental results obtained for searching on a protein sequence. Entries that state
> 200 indicate that the algorithm took longer than 200ms to search and timed out.
m

8

16

32

64

128

256

512

AOSOq
BNDMq
BSDMq
BXSq
EBOM
FSBq,s
KBNDM
LBNDM
SBNDMq

33.7(4)
19.18(2)
17.63(3)
15.51(2)
15.98
15.87(2,0)
45.47
68.43
15.83(2)

24.18(4)
11.96(2)
10.06(4)
9.95(3)
10.97
9.96(3,1)
25.63
42.74
10.85(2)

16.35(6)
8.36(4)
7.58(4)
7.77(4)
8.97
7.96(3,1)
14.9
20.28
8.36(4)

16.17(6)
8.17(4)
6.8(4)
7.76(4)
8.11
7.98(3,1)
11.95
14.2
8.32(4)

16.16(6)
8.16(4)
6.69(4)
7.76(4)
7.06
7.97(3,1)
10.95
11.8
8.33(4)

16.16(6)
8.18(4)
6.66(4)
7.77(3)
5.52
7.98(3,1)
11.42
9.89
8.33(4)

16.2(6)
8.19(4)
6.62(4)
7.79(4)
3.11
7.95(3,1)
11.41
9.21
8.33(4)

FJS
HASHq
IOM
WOM

69.82
80.9(3)
62.5
67.28

46.86
37.23(3)
41.85
43.7

35.53
19.5(3)
31.74
31.41

31.05
12.55(5)
27.41
25.62

28.51
10.79(5)
25.23
21.99

27.39
11.48(3)
24.26
19.92

27.22
12.9(3)
24.22
18.49

BRAMq
HCq,α
QFq,s
UFMq
WFRq

31.48(3)
16.17(3,11)
16.03(2,6)
23.15(3)
26.03(2)

16.36(3)
9.38(3,11)
9.93(3,4)
13.66(3)
12.36(4)

11.15(3)
7.58(3,11)
7.63(3,4)
9.69(7)
8.37(4)

8.58(7)
6.84(6,12)
6.84(4,3)
7.72(8)
7.2(4)

7.79(7)
6.23(3,11)
6.27(3,4)
7.68(8)
6.61(4)

6.81(7)
4.12(3,11)
4.16(3,4)
6.37(7)
4.75(4)

3.71(7)
2.37(4,12)
2.33(4,3)
3.54(8)
2.52(5)

SHCq,α
TWFRq

15.85(3,11)
23.37(4)

9.23(3,11)
10.73(4)

7.49(3,11)
8.09(4)

6.82(4,12)
7.1(4)

6.19(3,11)
6.52(4)

4.06(3,11)
4.62(4)

2.29(4,12)
2.49(5)

EPSM

11.44

10.06

6.87

6.69

5.62

3.52

1.95

SEA 2024


Table 3: Experimental results obtained for searching on an English text. Entries that state \( m > 200 \) indicate that the algorithm took longer than 200ms to search and timed out.

<table>
<thead>
<tr>
<th>m</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOSO</td>
<td>36.67 (4)</td>
<td>24.19 (4)</td>
<td>16.3 (6)</td>
<td>16.11 (6)</td>
<td>16.11 (6)</td>
<td>16.08 (6)</td>
<td>16.09 (6)</td>
</tr>
<tr>
<td>BNDM</td>
<td>24.57 (2)</td>
<td>13.37 (4)</td>
<td>8.9 (4)</td>
<td>8.83 (4)</td>
<td>8.79 (4)</td>
<td>8.83 (4)</td>
<td>8.76 (4)</td>
</tr>
<tr>
<td>BSDM</td>
<td>18.92 (2)</td>
<td>10.8 (4)</td>
<td>8.07 (4)</td>
<td>7.07 (6)</td>
<td>6.85 (6)</td>
<td>6.87 (6)</td>
<td>6.81 (4)</td>
</tr>
<tr>
<td>BXXS</td>
<td>21.18 (2)</td>
<td>11.92 (3)</td>
<td>8.33 (4)</td>
<td>8.31 (4)</td>
<td>8.31 (4)</td>
<td>8.36 (4)</td>
<td>8.29 (4)</td>
</tr>
<tr>
<td>FSB</td>
<td>20.41 (2)</td>
<td>11.97 (3.1)</td>
<td>8.71 (4.2)</td>
<td>8.71 (4.2)</td>
<td>8.7 (4.2)</td>
<td>8.75 (4.2)</td>
<td>8.68 (4.2)</td>
</tr>
<tr>
<td>KBNDM</td>
<td>49.08</td>
<td>30.41</td>
<td>19.88</td>
<td>15.23</td>
<td>11.89</td>
<td>11.82</td>
<td>11.81</td>
</tr>
<tr>
<td>LBNDM</td>
<td>81.15</td>
<td>49.51</td>
<td>26.67</td>
<td>16.95</td>
<td>13.08</td>
<td>10.47</td>
<td>9.13</td>
</tr>
<tr>
<td>SBNDM</td>
<td>20.75 (3)</td>
<td>13.41 (4)</td>
<td>8.89 (4)</td>
<td>8.84 (4)</td>
<td>8.83 (4)</td>
<td>8.9 (4)</td>
<td>8.83 (4)</td>
</tr>
<tr>
<td>FJS</td>
<td>75.05</td>
<td>49.48</td>
<td>36.19</td>
<td>29.83</td>
<td>24.97</td>
<td>21.69</td>
<td>20.46</td>
</tr>
<tr>
<td>HASH</td>
<td>80.30 (3)</td>
<td>36.76 (3)</td>
<td>18.67 (3)</td>
<td>12.33 (3)</td>
<td>10.71 (3)</td>
<td>11.49 (3)</td>
<td>13.87 (3)</td>
</tr>
<tr>
<td>IOM</td>
<td>67.84</td>
<td>45.18</td>
<td>33.21</td>
<td>26.8</td>
<td>22.51</td>
<td>19.83</td>
<td>19.18</td>
</tr>
<tr>
<td>WOM</td>
<td>72.35</td>
<td>45.86</td>
<td>31.92</td>
<td>25.4</td>
<td>20.48</td>
<td>17.87</td>
<td>16.75</td>
</tr>
<tr>
<td>BRAM</td>
<td>31.99 (3)</td>
<td>17.98 (3)</td>
<td>10.85 (5)</td>
<td>8.35 (5)</td>
<td>7.7 (5)</td>
<td>6.14 (5)</td>
<td>3.23 (7)</td>
</tr>
<tr>
<td>HC</td>
<td>17.52 (3,1)</td>
<td>10.67 (3,1)</td>
<td>8.08 (6,12)</td>
<td>6.84 (6,12)</td>
<td>6.49 (6,11)</td>
<td>4.57 (6,11)</td>
<td>2.58 (4,12)</td>
</tr>
<tr>
<td>QF</td>
<td>20.26 (3)</td>
<td>10.47 (3)</td>
<td>8.86 (3)</td>
<td>8.99 (3)</td>
<td>8.5 (3)</td>
<td>6.5 (3)</td>
<td>4.64 (3)</td>
</tr>
<tr>
<td>UF</td>
<td>24.77 (3)</td>
<td>13.94</td>
<td>9.02 (4)</td>
<td>7.4 (4)</td>
<td>7.4 (4)</td>
<td>5.69 (4)</td>
<td>3.21 (8)</td>
</tr>
<tr>
<td>WFR</td>
<td>26.47 (3)</td>
<td>12.75 (4)</td>
<td>8.67 (4)</td>
<td>7.3 (4)</td>
<td>6.63 (4)</td>
<td>4.77 (4)</td>
<td>2.54 (5)</td>
</tr>
<tr>
<td>SHC</td>
<td>17.14 (3,11)</td>
<td>10.41 (3,11)</td>
<td>8.0 (5,12)</td>
<td>7.0 (5,12)</td>
<td>6.44 (5,11)</td>
<td>4.52 (5,11)</td>
<td>2.48 (2,12)</td>
</tr>
<tr>
<td>TWFR</td>
<td>23.55 (4)</td>
<td>11.14 (4)</td>
<td>8.31 (4)</td>
<td>7.11 (4)</td>
<td>6.53 (4)</td>
<td>4.07 (4)</td>
<td>2.51 (5)</td>
</tr>
<tr>
<td>EPSM</td>
<td>12.83</td>
<td>9.95</td>
<td>6.81</td>
<td>6.62</td>
<td>5.56</td>
<td>3.47</td>
<td>1.94</td>
</tr>
</tbody>
</table>

For all alphabets and pattern lengths where \( m > 8 \), HC, \( q, \alpha \) and its variant SHC, \( q, \alpha \) consistently demonstrate superior speed. Specifically, on protein sequences alone, BSDM, \( q \) outperforms others in terms of speed when \( m = 64 \). On genome sequences, TWFR exhibits the fastest performance when \( m \geq 512 \), and for BSDM, \( m = 8 \). Notably, among the fastest algorithms for cases where \( m \geq 16 \), the predominant approach is based on weak factor recognition, whereas for instances with \( m < 16 \), automata-based algorithms typically emerge as the fastest.

Upon comparing SHC and HC, it becomes evident that SHC outpaces its counterpart, consistently securing the highest number of top-ranking results across all alphabets. With the exception of genome sequences, SHC stands out as the fastest algorithm for various pattern lengths. In instances where SHC is not the fastest, HC frequently claims the top position. Notably, when neither algorithm secures the fastest time, their respective timings typically fall within a range of 105% of the fastest recorded time.

Regarding the performances of the different variants of the HC algorithm, the optimal parameter value for \( \alpha \) has consistently demonstrated to be 12, with \( \alpha = 11 \) yielding the best running time only in occasional instances. Concerning the size of the \( q \)-grams utilized in the algorithm implementation, as intuitively anticipated, larger \( q \) values yield enhanced performance as the pattern size increases. Nevertheless, the parameter \( q \) should be consistently constrained within the range of 3 to 6.

If your primary concern is the count of pattern occurrences within the text rather than their specific positions, the EPSM algorithm stands out as a top performer. Leveraging the latest SSE (or AVX) instructions inherent in contemporary processors, it consistently delivers superior speed with markedly reduced execution times. Exploring the feasibility of implementing a weak factor recognition algorithm using similar SSE technologies could be an intriguing avenue for further investigation.
5 Conclusions

In this paper, we introduced the Hash Chain algorithm (HC) and its variant SHC, representing innovative approaches to the online exact string matching problem. These algorithms leverage weak factor recognition and a hashing function that connects adjacent hash values of the pattern $x$.

Our comprehensive experimental evaluation reveals the remarkable competitiveness of our proposed solutions when benchmarked against state-of-the-art algorithms documented in the literature. The impressive performance exhibited by HC and SHC, along with other analogous weak factor recognizers, underscores the promise of weak factor recognition as a compelling approach in the realm of pattern recognition. This success encourages and paves the way for further exploration and research in the same direction.

Furthermore, contemplating a linear version of HC and SHC, mirroring the conceptual framework of the Linear Weak Factor Recognition algorithm[8], appears to be a feasible and promising avenue for future investigations.

References


Convex Relaxation for the Generalized Maximum-Entropy Sampling Problem

Gabriel Ponte
University of Michigan, Ann Arbor, MI, USA

Marcia Fampa
Universidade Federal do Rio de Janeiro, Brazil

Jon Lee
University of Michigan, Ann Arbor, MI, USA

Abstract
The generalized maximum-entropy sampling problem (GMESP) is to select an order-$s$ principal submatrix from an order-$n$ covariance matrix, to maximize the product of its $t$ greatest eigenvalues, $0 < t \leq s < n$. It is a problem that specializes to two fundamental problems in statistical design theory: (i) maximum-entropy sampling problem (MESP); (ii) binary D-optimality (D-Opt). In the general case, it is motivated by a selection problem in the context of PCA (principal component analysis).

We introduce the first convex-optimization based relaxation for GMESP, study its behavior, compare it to an earlier spectral bound, and demonstrate its use in a branch-and-bound scheme. We find that such an approach is practical when $s - t$ is very small.

2012 ACM Subject Classification Theory of computation → Mixed discrete-continuous optimization; Mathematics of computing → Mathematical optimization

Keywords and phrases maximum-entropy sampling, D-optimality, convex relaxation, branch-and-bound, integer nonlinear optimization, principal component analysis

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.25


1 Introduction
Let $C$ be a symmetric positive-semidefinite matrix with rows/columns indexed from $N := \{1, 2, \ldots, n\}$, with $n > 1$ and rank($C$) := $r$. For integers $t$ and $s$, such that $0 < t \leq r$ and $t \leq s < n$, we define the generalized maximum-entropy sampling problem (see [25, 16])

$$z(C, s, t) := \max \left\{ \sum_{\ell=1}^{t} \log(\lambda_\ell(C[S(x), S(x)])) : e^T x = s, \quad x \in \{0, 1\}^n \right\}, \quad \text{(GMESP)}$$

where $S(x)$ denotes the support of $x \in \{0, 1\}^n$, $C[S, S]$ denotes the principal submatrix indexed by $S$, and $\lambda_\ell(X)$ denotes the $\ell$-th greatest eigenvalue of a symmetric matrix $X$.

Twenty-five years ago, GMESP was introduced as a common generalization of MESP and binary D-Opt (see [25], but not widely disseminated until [16]). MESP, a central problem in statistics and information theory, corresponds to the problem of selecting a subvector of size $s$ from a Gaussian $n$-vector, so as to maximize the “differential entropy” (see [23]) of
Convex Relaxation for GMESP

the chosen subvector; see [9]. MESP is the special case of GMESP for which \( t := s \). The relationship with binary D-Opt is more involved. Given an \( n \times r \) matrix \( A \) of full column rank, binary D-Opt corresponds to the special case of GMESP for which \( C := AA^T \), and \( t := r \). D-Opt is equivalent to the problem of selecting a set of \( s \) design points from a given set of \( n \) potential design points (the rows of \( A \)), so as to minimize the volume of a confidence ellipsoid for the least-squares parameter estimates in the resulting linear model (assuming additive Gaussian noise); see [21], for example.

For the general case of GMESP, we can see it as motivated by a selection problem in the context of PCA (principal component analysis); see, for example, [12] and the references therein, for the important topic of PCA. Specifically, GMESP amounts to selecting a subvector of size \( s \) from a Gaussian \( n \)-vector, so that geometric mean of the variances associated with the \( t \) largest principal components is maximized. Linking this back to MESP, we can see that problem as selecting a subvector of size \( s \) from a Gaussian \( n \)-vector, so that the geometric mean of the variances associated with all principal components is maximized. We use the geometric mean of the variances, so as to encourage a selection where all \( t \) of them are large and similar in value;\(^2\) we note that maximizing the geometric mean is equivalent to maximizing the log of the product.

Expanding on our motivation for GMESP, we assume that we are in a setting where we have \( n \) observable Gaussian random variables, with a possibly low-rank covariance matrix. We assume that observations are costly, and so we want to select \( s \ll n \) for observation. Even the \( s \) selected random variables may have a low-rank covariance matrix. Posterior to the selection, we would then carry out PCA on the associated order-\( s \) covariance matrix, with the aim of identifying the most informative \( t < s \) latent/hidden random variables, where we define most informative as corresponding to maximizing the geometric mean of the variances of the \( t \) dominant principal components.

We also define the constrained generalized maximum-entropy sampling problem

\[
 z(C, s, t, A, b) := \max \left\{ \sum_{\ell=1}^{t} \log(\lambda_{\ell}(C[S(x), S(x)])) : e^T x = s, \ Ax \leq b,\ x \in \{0,1\}^n \right\},
\]

(cGMESP)

which is useful in practical applications where there are budget constraints, logistical constraints, etc., on which sets of size \( s \) are feasible. Correspondingly, we also refer to CMESP (the constrained version of MESP) and CD-Opt (the constrained version of D-Opt).

The main approach for solving GMESP and cGMESP to optimality is B&B (branch-and-bound); see [16]. Lower bounds are calculated by local search (especially for GMESP), rounding, etc. Upper bounds are calculated in a variety of ways. The only upper-bounding method in the literature uses spectral information; see [25, 16].

Some very good upper-bounding methods for CMESP and CD-Opt are based on convex relaxations; see [4, 1, 2, 20, 19, 8, 21]. For CMESP, a “down branch” is realized by deleting a symmetric row/column pair from \( C \). An “up branch” corresponds to calculating a Schur complement. For CD-Opt, a “down branch” amounts to eliminating a row from \( A \), and an “up branch” corresponds to adding a rank-1 symmetric matrix before applying a determinant operator to an order-\( r \) symmetric matrix that is linear in \( x \) (see [18] and [21] for details).

\(^2\) In this spirit, the product of sample variances is used in Bartlett’s test of homogeneity of variances; see [24, Section 10.21, pp. 296].
For the general case of CGMESP and the spectral bounding technique, we refer to [16] for a discussion of an “up branch”, which is actually quite complicated and probably not very efficient. We will present a convex relaxation for CGMESP that is amenable to the use of a simple “up branch”. Our new “generalized factorization bound” for CGMESP generalizes (i) the “factorization bound” for CMESP (see [8]), and (ii) the “natural bound” for binary CD-Opt (see [21]). We wish to emphasize that it does not generalize the “factorization bound” for binary CD-Opt.

Organization and Contributions. In §2, we introduce the generalized factorization bound as the Lagrangian dual of a non-convex relaxation and establish its basic properties and its relation with the spectral bound from [16]. In §3, we apply Lagrangian duality again, reaching a more tractable formulation for calculating the generalized factorization bound. In §4, we present results from computational experiments with a B&B algorithm based on the generalized factorization bound, where we demonstrate favorable computational performance when $s - t$ is small. In §5, we describe some directions for further study.

Notation. We let $S^+_{+}$ (resp., $S^+_{++}$) denote the set of positive semidefinite (resp., definite) symmetric matrices of order $n$. We let $\text{Diag}(x)$ denote the $n \times n$ diagonal matrix with diagonal elements given by the components of $x \in \mathbb{R}^n$, and $\text{diag}(X)$ denote the $n$-dimensional vector with elements given by the diagonal elements of $X \in \mathbb{R}^{n \times n}$. We denote an all-ones vector by $\mathbf{e}$ and the $i$-th standard unit vector by $\mathbf{e}_i$. For matrices $A$ and $B$ with the compatible shapes, $A \bullet B := \text{tr}(A'B)$ is the matrix dot-product. For a matrix $A$, we denote row $i$ by $A_i$, and column $j$ by $A_j$.

2 Generalized factorization bound

Suppose that the rank of $C$ is $r \geq t$. We factorize $C = FF^t$, with $F \in \mathbb{R}^{n \times k}$, for some $k$ satisfying $r \leq k \leq n$. This could be a Cholesky-type factorization, as in [20] and [19], where $F$ is lower triangular and $k := r$, could be derived from the spectral decomposition $C = \sum_{i=1}^r \lambda_i v_i v_i^T$, by selecting $\sqrt{\lambda_i}$ as the column $i$ of $F$, $i = 1, \ldots, k = r$, or it could be derived from the matrix square root of $C$, where $F := C^{1/2}$, and $k := n$.

For $x \in [0, 1]^n$, we define $f(x) := \sum_{i=1}^r \log(\lambda_i(F(x)))$, where $F(x) := \sum_{i \in N} F_i^x F_i$, $x_i = F^T \text{Diag}(x) F$, and

$$z_{\text{GFact}}(C, s, t, A, b; F) := \max \{ f(x) : \mathbf{e}^T x = s, \ Ax \leq b, \ 0 \leq x \leq \mathbf{e} \}.$$  \hspace{1cm} (GFact)

Next, we see that GFact gives an upper bound for CGMESP.

\textbf{Theorem 1.}

$$z(C, s, t, A, b) \leq z_{\text{GFact}}(C, s, t, A, b; F).$$

\textbf{Proof.} It suffices to show that for any feasible solution $x$ of CGMESP with finite objective value, we have $\sum_{i=1}^r \log(\lambda_i(C[S(x), S(x)])) = f(x)$. Let $S := S(x)$. Then, for $S \subset N$ with $|S| = s$ and rank $C[S, S] \geq t$, we have $F(x) = \sum_{i \in S} F_i^x F_i = F[S, \cdot]^T F[S, \cdot] = S_+^S$. Also, we have $C[S, S] = F[S, \cdot]^T F[S, \cdot] \in S_+^S$. Now, we observe that the nonzero eigenvalues of $F[S, \cdot]^T F[S, \cdot]$ and $F[S, \cdot]^T F[S, \cdot] = S_+^S$ are identical and the rank of these matrices is at least $t$. So, the $t$ largest eigenvalues of these matrices are positive and identical. The result follows. \hfill \Box
From the proof, we see that replacing $0 \leq x \leq e$ with $x \in \{0,1\}^n$ in GFact, we get an exact mixed-integer nonlinear optimization (MINLO) formulation for CGMESP. But GFact is not generally a convex program, so we cannot make direct use of such a formulation and Theorem 1. We will overcome this difficulty using Lagrangian duality, obtaining an upper bound for $z_{\text{GFact}}$. We first re-cast GFact as

$$\max \left\{ \sum_{\ell=1}^t \log (\lambda_\ell (W)) : F(x) = W, \ e^T x = s; \ Ax \leq b; \ 0 \leq x \leq e \right\},$$

and consider the Lagrangian function

$$L(W,x,\Theta,v,\nu,\pi,\tau) := \sum_{\ell=1}^t \log (\lambda_\ell (W)) + \Theta \bullet (F(x) - W) + v^T x + \nu^T (e - x) + \pi^T (b - Ax) + \tau (s - e^T x),$$

with $\text{dom} \ L = \mathbb{S}_+^{k,t} \times \mathbb{R}^n \times \mathbb{S}_+^k \times \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}$, where $\mathbb{S}_+^{k,t}$ denotes the convex set of $k \times k$ positive semidefinite matrices with rank at least $t$.

The corresponding dual function is

$$L^*(\Theta,v,\nu,\pi,\tau) := \sup_{W \in \mathbb{S}_+^{k,t}} L(W,x,\Theta,v,\nu,\pi,\tau),$$

and the corresponding Lagrangian dual problem is

$$z_{\text{DGFact}}(C,s,t,A,b;F) := \inf \{ L^*(\Theta,v,\nu,\pi,\tau) : \nu \geq 0, \ \pi \geq 0 \}. $$

We call $z_{\text{DGFact}} := z_{\text{DGFact}}(C,s,t,A,b;F)$ the generalized factorization bound. We note that

$$\sup_{W \in \mathbb{S}_+^{k,t}} \left\{ \sum_{\ell=1}^t \log (\lambda_\ell (W)) + \Theta \bullet (F(x) - W) + v^T x + \nu^T (e - x) + \pi^T (b - Ax) + \tau (s - e^T x) \right\}$$

$$= \sup_{W \in \mathbb{S}_+^{k,t}} \left\{ \sum_{\ell=1}^t \log (\lambda_\ell (W)) - \Theta \bullet W \right\} + \sup_x \left\{ \Theta \bullet (F(x) + v^T x - \nu^T x - \pi^T Ax - \tau e^T x + \nu^T e + \pi^T b + \tau s) \right\}.$$ (1)

In Theorems 2 and 3 we analytically characterize the suprema in (1) and (2). The proof of Theorem 2 can be found in the full version [22]. The result in Theorem 3 follows from the fact that a linear function is bounded above only when it is identically zero.

**Theorem 2** (see [19], Lemma 1). For $\Theta \in \mathbb{S}_+^k$, we have

$$\sup_{W \in \mathbb{S}_+^{k,t}} \left( \sum_{\ell=1}^t \log (\lambda_\ell (W)) - W \bullet \Theta \right) = \begin{cases} -t - \sum_{\ell=k-t+1}^k \log (\lambda_\ell (\Theta)), & \text{if } \Theta > 0; \\ +\infty, & \text{otherwise}. \end{cases}$$ (3)

**Theorem 3.** For $(\Theta,v,\nu,\pi,\tau) \in \mathbb{S}_+^k \times \mathbb{R}^n \times \mathbb{S}_+^k \times \mathbb{R}^m \times \mathbb{R}$, we have

$$\sup_x \left( \Theta \bullet (F(x) + v^T x - \nu^T x - \pi^T Ax - \tau e^T x + \nu^T e + \pi^T b + \tau s) \right)$$

$$= \begin{cases} \nu^T e + \pi^T b + \tau s, & \text{if } \text{diag}(F \Theta F^T) + v - \nu - A^T \pi - \tau e = 0; \\ +\infty, & \text{otherwise}. \end{cases}$$

Considering Theorems 2 and 3, we see that the Lagrangian dual of GFact is equivalent to

$$z_{\text{DGFact}}(C,s,t,A,b;F) = \min \left\{ -\sum_{\ell=k-t+1}^k \log (\lambda_\ell (\Theta)) + \nu^T e + \pi^T b + \tau s - t \right\}$$

subject to:

$$\text{diag}(F \Theta F^T) + v - \nu - A^T \pi - \tau e = 0,$$

$$\Theta > 0, \ v \geq 0, \ \nu \geq 0, \ \pi \geq 0.$$ (DGFact)
From Lagrangian duality, we conclude that DGFact is a convex program. Nevertheless, we note that GFact is not generally a convex program, so we will not generally have strong duality between GFact and DGFact.

Next, we establish two properties for the generalized factorization bound that were similarly established for the factorization bound for MESP in [8]. Specifically, we show that the generalized factorization bound for CGMESP is invariant under multiplication of $C$ by a scale factor $\gamma$, up to the scale factor $-t \log \gamma$, and is also independent of the factorization of $C$. The proofs of the results are similar to the ones presented in [8] for MESP.

**Theorem 4.** For all $\gamma > 0$ and factorizations $C = FF^T$, we have

$$z_{\text{DGFact}}(C, s, t, A; b; F) = z_{\text{DGFact}}(\gamma C, s, t, A; \sqrt{\gamma} F) - t \log \gamma.$$ 

**Theorem 5.** $z_{\text{DGFact}}(C, s, t, A; b; F)$ does not depend on the chosen $F$.

[16] presents a spectral bound for GMESP, $\sum_{t=1}^{\ell} \log \lambda_t(C)$. Next, we present a relation between the generalized factorization bound and the spectral bound for GMESP.

**Theorem 6.** Let $C \in S^n$, with $r := \text{rank}(C) > 0$, and $t \leq s < n$. Then, for all factorizations $C = FF^T$, we have

$$z_{\text{DGFact}}(C, s, t; F) - \sum_{t=1}^{\ell} \log \lambda_t(C) \leq t \log \left( \frac{1}{\ell} \right).$$

**Proof.** Let $C = \sum_{t=1}^{r} \lambda_t(C) u_t u_t^T$ be a spectral decomposition of $C$. By Theorem 5, it suffices to take $F$ to be the symmetric matrix $\sum_{t=1}^{\ell} \sqrt{\lambda_t(C)} u_t u_t^T$.

We consider the solution for DGFact given by: $\hat{\Theta} := \frac{1}{s} \left( C^T + \frac{1}{\lambda_t(C)} (I - CC^T) \right)$, where $C^T := \sum_{t=1}^{r} \frac{1}{\lambda_t(C)} u_t u_t^T$ is the Moore-Penrose pseudoinverse of $C$, $\hat{\nu} := \frac{1}{s} e - \text{diag}(F \hat{\Theta} F^T)$, $\hat{\lambda} := 0$, $\hat{\pi} := 0$, and $t := \frac{1}{\ell}$. We can verify that the $r$ least eigenvalues of $\hat{\Theta}$ are $\frac{1}{s} \frac{1}{\lambda_t(C)}$, $\frac{1}{s} \frac{1}{\lambda_t(C)}$, ... and the $n-r$ greatest eigenvalues are all equal to $\frac{1}{s} \frac{1}{\lambda_t(C)}$. Therefore, $\hat{\Theta}$ is positive definite.

The equality constraint of DGFact is clearly satisfied at this solution. Additionally, we can verify that $F \hat{\Theta} F^T = \frac{1}{s} \sum_{t=1}^{\ell} u_t u_t^T$. As $\sum_{t=1}^{\ell} u_t u_t^T \leq I$, we conclude that $\text{diag}(F \hat{\Theta} F^T) \leq \frac{1}{s} e$. Therefore, $\hat{\nu} \geq 0$, and the solution constructed is a feasible solution to DGFact. Finally, we can see that the objective value of this solution is equal to the spectral bound added to $t \log(s/t)$. The result then follows. ▶

**Remark 7.** Considering Theorem 6, for $r = s - k$, with constant integer $k \geq 0$, $\lim_{s \to \infty} t \log(s/t) = k$. Therefore, in this limiting regime, the generalized factorization bound is no more than an additive constant worse than the spectral bound.

Considering Theorem 6, we will see that key quantities are discrete concave in $t$, in such a way that we get a concave upper bound that only depends on $t$ and $s$ for the difference of two discrete concave upper bounds (which depend on $C$ as well).

**Theorem 8.**

(a) $t \log \left( \frac{1}{\ell} \right)$ is (strictly) concave in $t$ on $R^+$

(b) $\sum_{t=1}^{\ell} \log \lambda_t(C)$ is discrete concave in $t$ on $\{1, 2, \ldots, r\}$

(c) $z_{\text{DGFact}}(C, s, t, A; b; F)$ is discrete concave in $t$ on $\{1, 2, \ldots, k\}$.

**Proof.** For (a), we see that the second derivative of the function is $-1/t$, which is negative on $R^+$. For (b), it is easy to check that discrete concavity is equivalent to $\lambda_t(C) \geq \lambda_{t+1}(C)$, for all integers $t$ satisfying $1 \leq t < r$, which we obviously have. For (c), first we observe that
we can view \( z_{DGFact}(C, s, t, A, b : F) \) as the pointwise minimum of \(-\sum_{\ell=k-t+1}^{k} \log (\lambda_{\ell}(\Theta)) + \nu^T e + \pi^T b + \tau s - t\) over the points in the convex feasible region of DGFact. So it suffices to demonstrate that the function \(-\sum_{\ell=k-t+1}^{k} \log (\lambda_{\ell}(\Theta)) + \nu^T e + \pi^T b + \tau s - t\) is discrete concave in \( t \). It is easy to check that this is equivalent to \( \lambda_{k-t+1}(C) \geq \lambda_{k-t+2}(C) \), for all integers \( t \) satisfying \( 1 < t \leq k \), which we obviously have.

Theorem 8, part (c) is very interesting, in connection with the motivating application to PCA. Using convexity, we can compute upper bounds on the value of changing the number \( t \) of dominant principal components considered, without having to actually solve further instances of DGFact (of course, solve those would give better upper bounds).

[16] extends the spectral bound to take advantage of the side constraints \( Ax \leq b \). In the full version [22], we extend Theorem 8, part (b), to that situation.

In the next theorem, we present for CGMESP, a key result to enhance the application of B&B algorithms to discrete optimization problems with convex relaxations. The principle described in the theorem is called variable fixing, and has been successfully applied to MESP (see [3, 4]). The similar proof of the theorem for MESP can be found in [9, Theorem 3.3.9].

\[ \text{Theorem 9. Let } \]
\[ \text{LB be the objective-function value of a feasible solution for CGMESP,} \]
\[ (\hat{\Theta}, \hat{\nu}, \hat{\pi}, \hat{\tau}) \text{ be a feasible solution for DGFact with objective-function value } \hat{\zeta}. \]
\[ \text{Then, for every optimal solution } x^* \text{ for CGMESP, we have:} \]
\[ x^*_j = 0, \quad \forall \ j \in N \text{ such that } \hat{\zeta} - \text{LB} < \hat{\nu}_j, \]
\[ x^*_j = 1, \quad \forall \ j \in N \text{ such that } \hat{\zeta} - \text{LB} < \hat{\nu}_j. \]

3 Duality for DGFact

Although it is possible to directly solve DGFact to calculate the generalized factorization bound, it is computationally more attractive to work the Lagrangian dual of DGFact. In the following, we construct this dual formulation.

Consider the Lagrangian function corresponding to DGFact, after eliminating the slack variable \( \nu \),
\[
\mathcal{L}(\Theta, \nu, \pi, \tau, x, y, w) := -\sum_{\ell=k-t+1}^{k} \log (\lambda_{\ell}(\Theta)) + \nu^T e + \pi^T b + \tau s - t + x^T (\text{diag}(F\Theta F^T) - \nu - A^T \pi - \tau e) - y^T \nu - w^T \pi,
\]
with \( \text{dom } \mathcal{L} = S^k_{++} \times \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^m \).

The corresponding dual function is
\[
\mathcal{L}^*(x, y, w) := \inf_{\Theta \in S^k_{++}, \nu, \pi, \tau} \mathcal{L}(\Theta, \nu, \pi, \tau, x, y, w), \tag{4}
\]
and the Lagrangian dual problem of DGFact is
\[
z_{DGFact}(C, s, t, A, b : F) := \max \{ \mathcal{L}^*(x, y, z) : x \geq 0, \ y \geq 0, \ w \geq 0 \}. \]

We note that DGFact has a strictly feasible solution (e.g., given by \( \hat{\Theta} := I, \hat{\nu} := 0, \hat{\nu} := \text{diag}(FF^T) = \text{diag}(C), \hat{\pi} := 0, \hat{\tau} := 0 \)). Then, Slater’s condition holds for DGFact and we are justified to use maximum in the formulation of the Lagrangian dual problem, rather than supremum, as the optimal value of the Lagrangian dual problem is attained.
We have that
\[ \inf_{\Theta \in S_k^{++}, \nu, \pi, \tau} L(\Theta, \nu, \pi, \tau, x, y, w) = \]
\[ \inf_{\Theta \in S_k^{++}} \left\{ -\sum_{t=k-t+1}^{k} \log (\lambda_t(\Theta)) + x^T \text{diag}(F \Theta F^T) - t \right\} \]
\[ + \inf_{\nu, \pi, \tau} \{ \nu^T(e - x - y) + \pi^T(b - Ax - w) + \tau(s - e^T x) \}. \]

Next, we discuss the infima in (5) and (6), for which the following lemma brings a fundamental result.

▶ Lemma 10 (see [20], Lemma 13). Let \( \lambda \in \mathbb{R}_+^k \) with \( \lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k \) and let \( 0 < t \leq k \). There exists a unique integer \( \iota \), with \( 0 \leq \iota < t \), such that
\[ \lambda_\iota > \frac{1}{t-\iota} \sum_{t=\iota+1}^{k} \lambda_t \geq \lambda_{\iota+1}, \]
with the convention \( \lambda_0 = +\infty \).

Suppose that \( \lambda \in \mathbb{R}_+^k \), and assume that \( \lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k \). Given integer \( t \) with \( 0 < t \leq k \), let \( \iota \) be the unique integer defined by Lemma 10. We define
\[ \phi_t(\lambda) := \sum_{\ell=1}^{\iota} \log (\lambda_\ell) + (t - \iota) \log \left( \frac{1}{t-\iota} \sum_{t=\iota+1}^{k} \lambda_t \right). \]

Also, for \( X \in S_k^{++} \), we define \( \Gamma_t(X) := \phi_t(\lambda(X)) \).

Considering the definition of \( \Gamma_t \), we analytically characterize the infimum in (5) in Theorem 11. Its proof can be found in the full version [22].

▶ Theorem 11 (see [20], Lemma 16). For \( x \in \mathbb{R}^n \), we have
\[ \inf_{\Theta \in S_k^{++}} -\sum_{t=k-t+1}^{k} \log (\lambda_t(\Theta)) + x^T \text{diag}(F \Theta F^T) - t \]
\[ = \left\{ \begin{array}{ll} \Gamma_t(F(x)) & \text{if } F(x) \succeq 0 \text{ and } \text{rank}(F(x)) \geq t; \\ -\infty & \text{otherwise}. \end{array} \right. \]

Finally, we analytically characterize the infimum in (6) in Theorem 12. Its proof follows from the fact that a linear function is bounded below only when it is identically zero.

▶ Theorem 12. For \( (x, y, w) \in \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^m \), we have
\[ \inf_{\nu, \pi, \tau} \nu^T(e - x - y) + \pi^T(b - Ax - w) + \tau(s - e^T x) \]
\[ = \left\{ \begin{array}{ll} 0 & \text{if } e - x - y = 0, \ b - Ax - w = 0, \ s - e^T x = 0; \\ -\infty & \text{otherwise}. \end{array} \right. \]

Considering Theorems 11 and 12, the Lagrangian dual of DGFact is equivalent to
\[ z_{\text{DDGFact}}(C, s, t, A, b; F) = \max \{ \Gamma_t(F(x)) : e^T x = s, \ Ax \leq b, \ 0 \leq x \leq e \}, \quad \text{(DDGFact)} \]

From Lagrangian duality, we have that DDGFact is a convex program. Moreover, because DGFact has a strictly feasible solution, if DDGFact has a feasible solution with finite objective value, then we have strong duality between DDGFact and DGFact.

Finally, for developing a nonlinear-programming algorithm for DDGFact, we consider in the next theorem, an expression for the gradient of its objective function. The proof is similar to the one presented for MESP in [8, Theorem 2.10].
\[ \frac{\partial}{\partial x_j} \Gamma_1(F(\hat{x})) = \sum_{t=1}^{r} \frac{1}{\lambda_t} (F_j(\hat{u}_t))^2 + \sum_{t=1}^{k} \frac{t-i}{\sum_{i=i+1}^{r} \lambda_i} (F_j(\hat{u}_t))^2. \]

As observed in [8], without the technical condition \( \frac{1}{r} \sum_{i=i+1}^{k} \lambda_t > \hat{\lambda}_{t+1} \), the formula above still give a subgradient of \( \Gamma_1 \) (see [19]).

As mentioned above, by replacing \( 0 \leq x \leq \mathbf{e} \) by \( x \in \{0,1\}^n \) in GFact, we get an exact but non-convex MINLO formulation for CGMESP. On the other hand, replacing \( 0 \leq x \leq \mathbf{e} \) by \( x \in \{0,1\}^n \) in DDGFact, we get a convex formulation for CGMESP, which is non-exact generally, except for the important case of \( t = s \) when it becomes exact. In Theorem 15, we present properties of the function \( \phi \) defined in (8), which show that the relaxation is non-exact for \( t < s \) and exact for \( t = s \). In Lemma 14 we prove the relevant facts for their understanding.

**Theorem 13.** Let \( F(\hat{x}) = \sum_{t=1}^{k} \hat{\lambda}_t u_t u_t^T \) be a spectral decomposition of \( F(\hat{x}) \). Let \( \hat{t} \) be the value of \( t \) in Lemma 10, where \( \lambda \) in Lemma 10 is \( \hat{\lambda} := \lambda(F(\hat{x})) \). If \( \frac{1}{r} \sum_{t=1}^{r} \hat{\lambda}_t > \hat{\lambda}_{t+1} \), then, for \( j = 1, 2, \ldots, n \),

\[ \Gamma_1(F(\hat{x})) \geq \sum_{t=1}^{r} \frac{1}{\lambda_t} F_j(\hat{u}_t)^2 + \sum_{t=1}^{k} \frac{t-i}{\sum_{i=i+1}^{r} \lambda_i} F_j(\hat{u}_t)^2. \]

**Proof.** Part (a) follows from:

\[ \Gamma_1(F(\hat{x})) \geq \sum_{t=1}^{r} \frac{1}{\lambda_t} F_j(\hat{u}_t)^2 + \sum_{t=1}^{k} \frac{t-i}{\sum_{i=i+1}^{r} \lambda_i} F_j(\hat{u}_t)^2. \]

Let \( \lambda \in \mathbb{R}^n_+ \) with \( \lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_r > \lambda_{r+1} = \cdots = \lambda_n = 0 \). Then,

(a) For \( t = r \), the \( i \) satisfying (7) is precisely \( \delta \).

(b) For \( t < s \), the \( i \) satisfying (7) is precisely \( r \).

**Proof.** For (a), the result follows because

\[ \frac{1}{r} \sum_{t=s+1}^{n} \lambda_t = \lambda_{s+1} \quad \text{and} \quad \lambda_s > \lambda_{s+1}. \]

For (b), the result follows because

\[ \frac{1}{r} \sum_{t=r+1}^{n} \lambda_t = 0 = \lambda_{r+1} \quad \text{and} \quad \lambda_r > \lambda_{r+1}. \]

**Theorem 15.** Let \( \lambda \in \mathbb{R}^n_+ \) with \( \lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_r > \lambda_{r+1} = \cdots = \lambda_n = 0 \). Then,

(a) \( \phi_t(\lambda) > \sum_{t=1}^{t} \log(\lambda_t) \), for \( 0 < t < r \),

(b) \( \phi_t(\lambda) = \sum_{t=1}^{t} \log(\lambda_t) \), for \( t = r \),

(c) \( \phi_t(\lambda) = -\infty \), for \( r < t \leq n \).

where we use \( \log(0) = -\infty \).

**Proof.** Part (a) follows from:

\[ (t-i) \log \left( \frac{1}{t-i} \sum_{t=i+1}^{n} \lambda_t \right) > (t-i) \log \left( \frac{1}{t-i} \sum_{t=i+1}^{r} \lambda_t \right) \geq (t-i) \left( \frac{\sum_{t=i+1}^{r} \log(\lambda_t)}{t-i} \right). \]

Parts (b) and (c) follow from Lemma 14.

**Remark 16.** We can conclude that \( z(C, s, t, A, b; F) \leq z_{DDGFact}(C, s, t, A, b; F) \) from the use of Lagrangian duality. But we note that Theorem 15 gives an alternative and direct proof for this result, besides showing in part (a), that the inequality is strict whenever the rank of any optimal submatrix \( C[S, S] \) for CGMESP is greater than \( t \).

Finally, we note that to apply the variable-fixing procedure described in Theorem 9 in a B&B algorithm to solve CGMESP, we need a feasible solution for DDGFact. To avoid wrong variable fixing by using dual information from near-optimal solutions to DDGFact, we will
show how to construct a rigorous feasible solution for DGFact from a feasible solution \( \hat{x} \) of DDGFact with finite objective value, with the goal of producing a small gap.

Although in CGMESP, the lower and upper bounds on the variables are zero and one, we will derive the construction of the dual solution considering the more general problem with lower and upper bounds on the variables given respectively by \( l \) and \( u \), that is, we consider the constraints \( l \leq x \leq u \) in DDGFact, instead of \( 0 \leq x \leq 1 \). The motivation for this, is to derive the technique to fix variables at any subproblem considered during the execution of the B&B algorithm, when some of the variables may already be fixed. Instead of redefining the problem with less variables in our numerical experiments, we found it more efficient to change the upper bound \( u_i \) from one to zero, when variable \( i \) is fixed at zero in a subproblem, and similarly, change the lower bound \( l_i \) from zero to one, when variable \( i \) is fixed at one.

We consider the spectral decomposition \( F(\hat{x}) = \sum_{\ell=1}^{n} \hat{\lambda}_{\ell} \hat{u}_{\ell} \hat{u}_{\ell}^T \), with \( \hat{\lambda}_{1} \geq \hat{\lambda}_{2} \geq \cdots \geq \hat{\lambda}_{r} > \hat{\lambda}_{r+1} = \cdots = \hat{\lambda}_{n} = 0 \). Notice that \( \text{rank}(F(\hat{x})) = \hat{r} \geq t \). Following [20], we set \( \hat{\Theta} := \sum_{\ell=1}^{k} \hat{\beta}_{\ell} \hat{u}_{\ell} \hat{u}_{\ell}^T \), where

\[
\hat{\beta}_{\ell} := \begin{cases}
1/\hat{\lambda}_{\ell}, & 1 \leq \ell \leq \hat{r}; \\
1/\hat{\delta}, & \hat{r} < \ell \leq \hat{r}; \\
(1 + \epsilon)/\hat{\delta}, & \hat{r} < \ell \leq k,
\end{cases}
\]

for any \( \epsilon > 0 \), where \( \hat{r} \) is the unique integer defined in Lemma 10 for \( \lambda_{\ell} = \hat{\lambda}_{\ell} \), and \( \hat{\delta} := \frac{1}{1 + \epsilon} \sum_{\ell=1}^{k} \hat{\lambda}_{\ell} \). From Lemma 10, we have that \( \hat{r} < t \). Then,

\[
-\sum_{\ell=1}^{t} \log(\hat{\beta}_{\ell}) = \sum_{\ell=1}^{t} \log \left( \hat{\lambda}_{\ell} \right) + (t - \hat{r}) \log(\hat{\delta}) = \Gamma_{t}(F(\hat{x})).
\]

Therefore, the minimum duality gap between \( \hat{x} \) in DDGFact and feasible solutions of DGFact of the form \( (\hat{\Theta}, v, \nu, \pi, \tau) \) is the optimal value of

\[
\begin{align*}
\min & -v^T l + \nu^T u + \pi^T b + \tau s - t \\
\text{subject to:} & v - \nu - A^T \pi - \tau e = -\text{diag}(F(\hat{\Theta}F^\pi)), \\
& v \geq 0, \nu \geq 0, \pi \geq 0.
\end{align*}
\]

We note that \( G(\hat{\Theta}) \) is always feasible (e.g., \( v := 0, \nu := \text{diag}(F(\hat{\Theta}F^\pi)), \pi := 0, \tau := 0 \)). Also, \( G(\hat{\Theta}) \) has a simple closed-form solution for GMESP, that is when there are no \( Ax \leq b \) constraints. To construct this optimal solution, we consider the permutation \( \sigma \) of the indices in \( N \), such that \( \text{diag}(F(\hat{\Theta}F^\pi))_{\sigma(1)} \geq \cdots \geq \text{diag}(F(\hat{\Theta}F^\pi))_{\sigma(n)} \). If \( u_{\sigma(1)} + \sum_{i=2}^{n} l_{\sigma(i)} > s \), we let \( \varphi := 0 \), otherwise we let \( \varphi := \max \{ j \in N : \sum_{i=1}^{j} u_{\sigma(i)} + \sum_{i=j+1}^{n} l_{\sigma(i)} \leq s \} \). We define \( P := \{ \sigma(1), \ldots, \sigma(\varphi) \} \) and \( Q := \{ \sigma(\varphi + 2), \ldots, \sigma(n) \} \). Then, we can verify that the following solution is optimal for \( G(\hat{\Theta}) \) when there are no side constraints (see the full version [22] for the proof of optimality of the solution).

\[
\begin{align*}
\tau^* := & \text{diag}(F(\hat{\Theta}F^\pi))_{\sigma(\varphi+1)}; \\
\nu_{\ell}^* := & \begin{cases}
\text{diag}(F(\hat{\Theta}F^\pi))_{\ell} - \tau^*, & \text{for } \ell \in P; \\
0, & \text{otherwise},
\end{cases} \\
v_{\ell}^* := & \begin{cases}
\tau^* - \text{diag}(F(\hat{\Theta}F^\pi))_{\ell}, & \text{for } \ell \in Q; \\
0, & \text{otherwise}.
\end{cases}
\end{align*}
\]

Although we are not able to prove the following conjecture concerning the dual solution for DGFact constructed as described above, it was supported by our numerical experiments. The conjecture is open even for the special cases of MESP and binary D-Opt.
25:10 Convex Relaxation for GMESP

- **Conjecture 17.** Considering GMESP, that is, considering the case when there are no $Ax \leq b$ constraints, if $\hat{x}$ is an optimal solution to DDGFact, then $(\hat{\Theta}, v^*, \nu^*, \tau^*)$ is an optimal solution to DGFact.

4 Experiments

Our initial experiments are only for GMESP. For all instances, we use a benchmark covariance matrix of dimension $n = 63$, originally obtained from J. Zidek (University of British Columbia), coming from an application for re-designing an environmental monitoring network; see [10] and [11]. This matrix has been used extensively in testing and developing algorithms for MESP; see [13, 15, 4, 17, 11, 5, 6, 1, 2, 7, 8].

4.1 Lower bounds

To get an idea of the performance of upper bounds, we wish to present gaps to good lower bounds. For good lower bounds for GMESP, we carry out an appropriate local search, in the spirit of [13, Sec. 4], starting from various good feasible solutions. Our local search is classical: Starting from some $S$ with $|S| = s$, we iteratively replace $S$ with $S + j - i$ when $\prod_{t=1}^s \lambda_t(C[S + j - i, S + j - i]) > \prod_{t=1}^s \lambda_t(C[S, S])$. We return at the end $x(S)$, the characteristic vector of $S$.

We have three methods for generating initial solutions for the local search.

- **Rounding a continuous solution.** Let $\lambda_t(C)$ be the $t$-th greatest eigenvalue of $C$, and $u_t$ be the corresponding eigenvector, normalized to have Euclidean length 1. We define $\tilde{x} \in \mathbb{R}^n$ by $\tilde{x}_j := \sum_{t=1}^s u_{t,j}^2$. It is easy to check (similar to [15, Sec. 3]) that $0 \leq \tilde{x} \leq e$ and $e^T \tilde{x} = t$. Next, we simply choose $S$ to comprise the indices $j$ corresponding to the $s$ biggest $\tilde{x}_j$; we note that this rounding method can be adapted to CGMESP, by instead solving a small integer linear optimization problem (see [15, Sec. 4]).

- **Greedy.** Starting from $S := \emptyset$, we identify the element $j \in N \setminus S$ that maximizes the product of the $\min\{t, |S| + 1\}$ biggest eigenvalues of $C[S + j, S + j]$. We let $S := S + j$, and we repeat while $|S| < s$.

- **Dual greedy.** Starting from $S := N$, we identify the element $j \in S$ that maximizes the product of the $t$ biggest eigenvalues of $C[S - j, S - j]$. We let $S := S - j$, and we repeat while $|S| > s$.

4.2 Behavior of upper bounds

To analyze the generalized factorization bound for GMESP and compare it to the spectral bound, we conducted two experiments using our covariance matrix with $n = 63$. In the first experiment, for each integer $k$ from 0 to 3, we consider the instances obtained when we vary $s$ from $k + 1$ to 61, and set $t := s - k$. In Figure 1, we depict for each $k$, the gaps given by the difference between the lower bounds for GMESP computed as described in §4.1, and the generalized factorization bound and the spectral bound for each pair $(s, t := s - k)$. We also depict the upper bound on the gap for the generalized factorization bound determined in Theorem 6, specifically given by the gap for the spectral bound added to $t \log(s/t)$. When $k = 0$ (instances of MESP), $t \log(s/t)$ is zero, confirming that the generalized factorization bound dominates the spectral bound as already proved in [8]. When $k$ increases, the generalized factorization bound becomes weaker and gets worse than the spectral bound when $s$ and $t$ get large enough. Nevertheless, the generalized factorization bound is still much stronger than the spectral bound for most of the instances considered, and we see that...
the upper bound on it given in Theorem 6 is, in general, very loose, specially for $s \in [10, 20]$. The variation on the gaps for generalized factorization bounds for the different values of $s$ is very small when compared to the spectral bound, showing its robustness.

![Figure 1](image1.png)

(a) $k = 0$.  
(b) $k = 1$.  
(c) $k = 2$.  
(d) $k = 3$.

**Figure 1** Gaps, varying $t = s - k$ ($n = 63$).

In the second experiment, for $s = 20$ and $s = 40$, we consider the instances obtained when we vary $t$ from 1 to $s$. Similarly to what we show in Figure 1, in Figure 2 we show the gaps for each instance, and the upper bound on the gap corresponding to the generalized factorization bound.

![Figure 2](image2.png)

(a) $s = 20$.  
(b) $s = 40$.

**Figure 2** Gaps, varying $t$, with $s$ fixed ($n = 63$).

In both plots in Figure 2, we see that after a certain value of $t$, the generalized factorization bound becomes stronger than the spectral bound. The interval for $t$ on which the generalized factorization bound is stronger is bigger when $s = 20$. The last two observations are expected. We note that from the formulae to calculate the bounds, the spectral bound does not take $s$ into account, so it should become worse as $s$ becomes smaller, compared to the dimension of $C$, and the generalized factorization bound take into account all the $s$ eigenvalues of the submatrix of $C$, so it should become worse as $t$ becomes smaller, compared to $s$. When $t = s$,
we see again in both plots that the generalized factorization bound dominates the spectral bound. We observe that the bound on the gap for the generalized factorization bound is always very loose for the values of $s$ considered. The plots confirm the analysis in Figure 1, showing now, for fixed $s$, that the generalized factorization bound becomes more promising when the difference between $t$ and $s$ is small.

4.3 Branch-and-bound

We coded a B&B algorithm using the generalized factorization bound and initializing it with the lower bound computed as described in §4.1. Branching is the standard fixing of a variable at 0 or 1 for the two child subproblems. We never create child subproblems if there is a unique feasible solution of a subproblem relaxation. For GMESP (no side constraints), such a solution will always be integer; for CGMESP, details on how to handle this are in [4, Sec. 4].

Because DDGFact is not an exact relaxation, we have to make some accommodations that are not completely standard for B&B. When we get an integer optimum solution $\bar{x}$ for a relaxed subproblem: (i) if the objective value of the relaxation of the subproblem being handled is above the lower bound, then we have to create child subproblems; (ii) we have to evaluate the true objective value of $\bar{x}$ in CGMESP, to determine if we should increase the lower bound.

We ran our B&B experiments on a 16-core machine (running Windows Server 2016 Standard): two Intel Xeon CPU E5-2667 v4 processors running at 3.20GHz, with 8 cores each, and 128 GB of memory. We coded our algorithms in Julia v.1.9.0. To solve the convex relaxation DDGFact, we apply Knitro v0.13.2. To solve GMESP, we employ the B&B algorithm in Juniper [14].

In Table 1, we present statistics for the B&B applied to instances where $C$ is the leading principal submatrix of order 32 of our 63-dimensional covariance matrix and $s$ varies from 2 to 31. For each $s$, we solve an instance of GMESP with $t = s - 1$ and an instance of MESP ($t = s$). For all instances, the initial lower bounds computed were optimal, so the B&B worked on proving optimality by decreasing the upper bound. In the first column of Table 1, we show $s$, and in the other columns, we show the following statistics for GMESP and MESP: the initial gap given by difference between the upper and lower bounds at the root node (root gap), the number of convex relaxations solved (nodes), the number of nodes pruned by bound (pruned bound), the number of variables fixed at 0 (1) by the procedure described in Theorem 9 (var fix 0 (1)), and the elapsed time (in seconds) for the B&B (B&B time). We observe that the difficulty of the problem significantly increases when $t$ becomes smaller than $s$ as the upper bounds become weaker, confirming the analysis of Figure 1. Nevertheless, the largest root gap for GMESP is about one, and we can solve all the instances in a time limit of 36,000 seconds. The quality of the generalized factorization bound for these instances of GMESP can be evaluated by the number of nodes pruned by bound in the B&B. For the most difficult instances (that took more than 9,000 seconds to be solved), about 20% of the nodes were pruned by bound. Moreover, we see that the generalized factorization bound led to an effective application of the variable-fixing procedure described in Theorem 9.

5 Outlook

We are left with some clear challenges. A key one is to obtain better upper bounds when $s - t$ is large, in hopes of exactly solving GMESP instances by B&B in such cases. In connection with this, we would like a bound that provably dominates the spectral bound (when $t < s$), improving on what we established with Theorem 6.
Table 1 Results for B&B with variable fixing.

<table>
<thead>
<tr>
<th>s</th>
<th>root gap</th>
<th>GMESP (t := s − 1)</th>
<th>MESP (t := s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>nodes</td>
<td>pruned bound var fix 0</td>
<td>time</td>
</tr>
<tr>
<td>2</td>
<td>0.362  186</td>
<td>1</td>
<td>75</td>
</tr>
<tr>
<td>3</td>
<td>0.481  556</td>
<td>14</td>
<td>269</td>
</tr>
<tr>
<td>4</td>
<td>0.594  2048</td>
<td>97</td>
<td>1048</td>
</tr>
<tr>
<td>5</td>
<td>0.699  7385</td>
<td>578</td>
<td>2755</td>
</tr>
<tr>
<td>6</td>
<td>0.731  20766</td>
<td>2183</td>
<td>6326</td>
</tr>
<tr>
<td>7</td>
<td>0.764  44714</td>
<td>5617</td>
<td>13473</td>
</tr>
<tr>
<td>8</td>
<td>0.772  69698</td>
<td>10267</td>
<td>21241</td>
</tr>
<tr>
<td>9</td>
<td>0.819  122251</td>
<td>22139</td>
<td>37011</td>
</tr>
<tr>
<td>10</td>
<td>0.821  163172</td>
<td>36421</td>
<td>54719</td>
</tr>
<tr>
<td>11</td>
<td>0.766  153960</td>
<td>39114</td>
<td>60981</td>
</tr>
<tr>
<td>12</td>
<td>0.718  165197</td>
<td>45035</td>
<td>71148</td>
</tr>
<tr>
<td>13</td>
<td>0.643  149194</td>
<td>40965</td>
<td>68837</td>
</tr>
<tr>
<td>14</td>
<td>0.676  296992</td>
<td>74606</td>
<td>129539</td>
</tr>
<tr>
<td>15</td>
<td>0.687  395037</td>
<td>96495</td>
<td>172151</td>
</tr>
<tr>
<td>16</td>
<td>0.691  428858</td>
<td>100016</td>
<td>178025</td>
</tr>
<tr>
<td>17</td>
<td>0.730  527963</td>
<td>115637</td>
<td>199472</td>
</tr>
<tr>
<td>18</td>
<td>0.755  549795</td>
<td>115105</td>
<td>187981</td>
</tr>
<tr>
<td>19</td>
<td>0.825  718108</td>
<td>139610</td>
<td>209843</td>
</tr>
<tr>
<td>20</td>
<td>0.862  698011</td>
<td>126972</td>
<td>172900</td>
</tr>
<tr>
<td>21</td>
<td>0.899  639993</td>
<td>109147</td>
<td>131023</td>
</tr>
<tr>
<td>22</td>
<td>0.946  588140</td>
<td>93020</td>
<td>96706</td>
</tr>
<tr>
<td>23</td>
<td>0.992  515171</td>
<td>75387</td>
<td>64191</td>
</tr>
<tr>
<td>24</td>
<td>1.014  369489</td>
<td>51736</td>
<td>35100</td>
</tr>
<tr>
<td>25</td>
<td>1.088  215042</td>
<td>28289</td>
<td>15379</td>
</tr>
<tr>
<td>26</td>
<td>0.965  98765</td>
<td>12020</td>
<td>5083</td>
</tr>
<tr>
<td>27</td>
<td>0.951  43052</td>
<td>4795</td>
<td>1410</td>
</tr>
<tr>
<td>28</td>
<td>0.912  13854</td>
<td>1368</td>
<td>234</td>
</tr>
<tr>
<td>29</td>
<td>0.900  3404</td>
<td>282</td>
<td>20</td>
</tr>
<tr>
<td>30</td>
<td>0.873  464</td>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td>31</td>
<td>0.832  32</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

References


Scalable Hard Instances for Independent Set Reconfiguration

Takehide Soh
Information Infrastructure and Digital Transformation Initiatives Headquarters, Kobe University, Japan

Takumu Watanabe
Graduate School of Information Sciences, Tohoku University, Japan

Jun Kawahara
Graduate School of Informatics, Kyoto University, Japan

Akira Suzuki
Graduate School of Information Sciences, Tohoku University, Japan

Takehiro Ito
Graduate School of Information Sciences, Tohoku University, Japan

Abstract

The Token Jumping problem, also known as the independent set reconfiguration problem under the token jumping model, is defined as follows: Given a graph and two same-sized independent sets, determine whether one can be transformed into the other via a sequence of independent sets. Token Jumping has been extensively studied, mainly from the viewpoint of algorithmic theory, but its practical study has just begun. To develop a practically good solver, it is important to construct benchmark datasets that are scalable and hard. Here, “scalable” means the ability to change the scale of the instance while maintaining its characteristics by adjusting the given parameters; and “hard” means that the instance can become so difficult that it cannot be solved within a practical time frame by a solver. In this paper, we propose four types of instance series for Token Jumping. Our instance series is scalable in the sense that instance scales are controlled by the number of vertices. To establish their hardness, we focus on the numbers of transformation steps; our instance series requires exponential numbers of steps with respect to the number of vertices. Interestingly, three types of instance series are constructed by importing theories developed by algorithmic research. We experimentally evaluate the scalability and hardness of the proposed instance series, using the SAT solver and award-winning solvers of the international competition for Token Jumping.

2012 ACM Subject Classification Computing methodologies → Discrete space search

Keywords and phrases Combinatorial reconfiguration, Benchmark dataset, Graph Algorithm, PSPACE-complete

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.26

Supplementary Material Dataset (Benchmark Instances, Solvers and Logs): https://github.com/TakehideSoh/Scalable-Hard-ISR

archived at swh:1:dir:c7100376b39143cf94f55312233283a4ced13440

Funding Takehide Soh: JSPS KAKENHI Grant Numbers JP21K11828, JP22K11973, JP23K11047
Akira Suzuki: JSPS KAKENHI Grant Numbers JP20K11666, JP20H05794
Takehiro Ito: JSPS KAKENHI Grant Numbers JP19K11814, JP20H05793

© Takehide Soh, Takumu Watanabe, Jun Kawahara, Akira Suzuki, and Takehiro Ito; licensed under Creative Commons License CC-BY 4.0

22nd International Symposium on Experimental Algorithms (SEA 2024),
Editor: Leo Liberti; Article No. 26; pp. 26:1–26:15
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
1 Introduction

Combinatorial reconfiguration [22, 37, 21] is a family of problems that involve finding a step-by-step transformation between two given feasible solutions of a combinatorial (search) problem such that all intermediate solutions are also feasible and each step respects a prescribed reconfiguration rule. One of the most well-studied reconfiguration problems is the Token Jumping problem, which is also known as the Independent Set Reconfiguration problem under the token jumping model [28]. Recall that an independent set $I$ of a graph $G$ is a vertex subset of $G$ such that no two vertices are adjacent in $G$. Imagine that a token is placed on each vertex in $I$. Given two independent sets (token placements) of $G$, we are asked whether there is a transformation from one into the other by moving a single token at a time while always maintaining independent sets of $G$. (See Figure 1 for a yes-instance, and Figure 2 for a no-instance.) Token Jumping has been extensively studied from the viewpoint of algorithmic theory (see the surveys in [11, 37]), and is known to be PSPACE-complete [28]. Since $\text{NP} \subseteq \text{PSPACE}$, this implies that there is a yes-instance such that even a shortest transformation requires a super-polynomial number of steps under the assumption of $\text{NP} \neq \text{PSPACE}$; otherwise, we can use a transformation (of polynomial steps) as the witness. Indeed, examples that actually require exponential numbers of steps have been constructed for some reconfiguration problems [17, 27, 9]. In the literature, standard algorithmic techniques designed for NP-complete problems are rarely extended to reconfiguration problems, and significant algorithmic developments are required. We think this is one of the main reasons why Token Jumping and other combinatorial reconfiguration problems have attracted much attention from the theoretical algorithms research community.

Token Jumping has been used to prove the PSPACE-completeness of several other reconfiguration problems. In this sense, Token Jumping is a theoretically central problem and is thus important, similar to the SAT problem for NP-complete problems. Indeed, in
addition to the theoretical studies, recently, there has been a growing focus on practical studies for Token Jumping. In particular, initiated by the international solver competitions for Token Jumping held in 2022 \(^1\) and 2023 \(^2\), named CoRe Challenge, general-purpose solvers for Token Jumping are studied. Christen et al. \([13]\) developed a solver that utilizes AI planning methods \([16]\). Yamada et al. \([42]\) proposed a solver based on answer set programming \([3]\). Ito et al. \([23]\) proposed a solver based on a data structure for a family of sets, called zero-suppressed binary decision diagram \([32]\). Notice that all the studies mentioned above are published in 2023 and 2024.

To accelerate these practical studies, it is crucial to construct benchmark datasets that are scalable and hard. Here, “scalable” means the ability to change the scale of the instance while maintaining its characteristics by adjusting the given parameters; and “hard” means that the instance can become so difficult that it cannot be solved within a practical time frame by a solver. To develop general-purpose solvers that do not rely on prior problem-specific knowledge, these properties are desirable for benchmark instances, as we have seen in history, e.g., for instances for SAT solvers. For example, the tower of Hanoi is easy to solve if we use problem-specific knowledge; however, interestingly, it is recognized and often used as a scalable and hard benchmark instance for general-purpose solvers such as SAT/CSP solvers (refer to CSP/SAT benchmark database \(^3\), \(^4\) and the literature \([36]\)).

It is important for solver developments to construct datasets reflecting the characteristic property of combinatorial reconfiguration, which gives a clear difference from NP-complete problems. To construct such datasets for Token Jumping, we focus on the numbers of required transformation steps. This is because, as mentioned above, one of the characteristic properties of Token Jumping is that there is a yes-instance such that even a shortest transformation requires a super-polynomial number of steps under the assumption of \(\text{NP} \neq \text{PSPACE}\). On the other hand, interestingly, Token Jumping is solvable in polynomial time if the number of transformation steps is bounded by a constant \([35]\). We thus think that the number of transformation steps gives a strong influence on the “hardness” of instances.

One may think that it is not so difficult to construct datasets for Token Jumping, because there are various graphs that can be used from publicly available datasets, e.g., the DIMACS Challenge \([25]\). However, from the results of CoRe Challenge 2022 \(^5\), despite the wide range of vertex numbers, from 11 to 10000, such instances do not need many transformation steps: instances with less than 10 steps occur in 80% of all DIMACS instances, and the longest step is only 112. Indeed, those instances were often easily solved by several solvers (including the SAT solver for an NP-complete problem).

In this paper, we propose four types of instance series for Token Jumping. Our instance series is scalable in the sense that instance scales are controlled by the number of vertices. We establish their hardness by ensuring that they require exponential numbers of steps with respect to the number of vertices. Interestingly, we constructed three of them by importing theories \([17, 27, 9]\) developed by algorithmic research (and the remaining one is from scratch). We experimentally evaluate the scalability and hardness of the proposed instance series using the SAT solver and award-winning solvers of the international competition CoRe Challenge \([39]\). By comparing with randomly generated instances, we will confirm that the proposed instances are scalable and hard enough.

---

5. https://core-challenge.github.io/2022result/
The paper is organized as follows. Section 2 introduces terminology used in the following sections. Section 3 explains how to create the proposed Token Jumping instance series. Section 4 explains three kinds of Token Jumping solvers with distinct characteristics. Section 5 shows empirical experiments demonstrating the scalability and hardness of the proposed instance series. Section 6 concludes this paper.

2 Preliminaries

In this section, we formally define the Token Jumping problem and its related terminologies.

Let $G = (V, E)$ be an unweighted and undirected graph. A vertex subset $I \subseteq V$ is an independent set of $G$ if no two vertices in $I$ are adjacent in $G$. For a positive integer $k$, the solution space $S_k(G)$ is a graph such that each node in $S_k(G)$ is an independent set $I$ of $G$ with $|I| = k$ and two nodes are joined by an edge in $S_k(G)$ if and only if the corresponding independent sets $I$ and $I'$ satisfy $|I \setminus I'| = |I' \setminus I| = 1$. A path in $S_k(G)$ connecting two nodes $I$ and $I'$ is called a reconfiguration sequence between $I$ and $I'$. The length of the reconfiguration sequence is defined to be the number of edges in the path.

Given a graph $G$ and two independent sets $I_s$ and $I_t$ of $G$ such that $|I_s| = |I_t| = k$, the Token Jumping problem is to determine whether or not there exists a reconfiguration sequence between $I_s$ and $I_t$ in $S_k(G)$. Note that Token Jumping is a decision problem, and hence we are not asked for an actual reconfiguration sequence as an output. Throughout this paper, we denote by $(G, I_s, I_t)$ an instance of Token Jumping.

To evaluate the “hardness” of instances of Token Jumping, we introduce the notion of the “distance” of an instance. Let $(G, I_s, I_t)$ be an instance of Token Jumping. Then, the distance of $(G, I_s, I_t)$ is defined as the shortest length of any reconfiguration sequence between $I_s$ and $I_t$; it is defined as $+\infty$ if the answer to $(G, I_s, I_t)$ is no. We say that an instance of Token Jumping is hard if its distance is exponential in the input size.

3 Scalable Hard Instances

In this section, we give four types of hard instance series for Token Jumping: three series are given in Section 3.1 by reductions from well-studied reconfiguration problems; and the last one in Section 3.2 is our original.

3.1 Hard instances based on reductions

In this subsection, we provide three types of hard instance series for Token Jumping, by introducing polynomial-time reductions from three well-studied reconfiguration problems. We will show (in Theorems 1, 2, and 3) that the distances of the corresponding instances of Token Jumping are at least those of source instances of the reductions. Then, because hard instance series are known for the three source reconfiguration problems of the reductions, we will obtain hard instance series for Token Jumping.

3.1.1 SAT-based series

Gopalan et al. [17] introduced the SAT Reconfiguration problem, as follows: we are given two satisfying truth assignments for a CNF formula $\phi$, and are asked to determine whether or not we can transform one into the other by flipping a truth assignment of a single variable at a time so that all intermediate results remain satisfying truth assignments for $\phi$. This problem is PSPACE-complete [17], and the complexities of SAT Reconfiguration and its related problems have been studied very precisely [17, 30, 31, 34]. Ito et al. [22, Theorem 2] gave the following reduction.
Theorem 1 ([22]). Let \( \phi \) be a 3-CNF-formula with \( N \) variables and \( M \) clauses, as an instance of SAT RECONFIGURATION. Then, there is a corresponding instance of Token Jumping such that the input size is \( O(NM) \) and its distance is at least that of the original instance.

We note in passing that the distance of the corresponding instance of Token Jumping is at most \( M \) times that of the original instance of SAT RECONFIGURATION.

Gopalan et al. [17, Lemma 3.7] constructed an instance series for SAT RECONFIGURATION such that formulas are 3-CNF and the distances of the instances are exponential in the input sizes. Then, by taking such SAT RECONFIGURATION instances, Theorem 1 gives a hard instance series for Token Jumping, which we call the SAT series in this paper.

3.1.2 Shortest-Path-based series

Kamiński et al. [27] introduced the Shortest Path RECONFIGURATION problem, as follows: we are given two shortest paths connecting two specified vertices \( s \) and \( t \) in an unweighted graph, and are asked to determine whether or not we can transform one into the other by exchanging a single vertex in a shortest path at a time so that all intermediate results remain shortest paths connecting \( s \) and \( t \). Surprisingly, this problem is PSPACE-complete [7, 41], and polynomial-time algorithms have been developed for restricted graph classes [1, 2, 7, 8, 15]. Kamiński et al. [28, Theorem 3] gave the following reduction.

Theorem 2 ([28]). Let \( G' \) be a graph with \( N \) vertices, as an instance of Shortest Path RECONFIGURATION. Then, there is a corresponding instance of Token Jumping such that the input size is \( O(N^2) \) and its distance is equal to that of the original instance.

Kamiński et al. [27, Theorem 1] constructed an instance series for Shortest Path RECONFIGURATION such that the distances of the instances are exponential in the input sizes. Then, by taking such Shortest Path RECONFIGURATION instances, Theorem 2 gives a hard instance series for Token Jumping, which we call the SP series in this paper.

3.1.3 List-Coloring-based series

Let \( C \) be a set of \( k \) colors. For a graph \( G' = (V', E') \), assume that each vertex \( v \) in \( V' \) has a list \( L(v) \subseteq C \) of colors. Then, a list coloring of \( G \) is to assign a color in \( L(v) \) to each vertex \( v \) in \( V' \) so that no two adjacent vertices receive the same color. Then, the List Coloring RECONFIGURATION problem is defined as follows: we are given two list colorings of \( G \), and are asked to determine whether or not we can transform one into the other by recoloring a single vertex at a time, so that all intermediate results remain list colorings of \( G \). This problem and its related problems appear in several research fields, such as Glauber dynamics in statistical physics (e.g., see [24]).

List Coloring RECONFIGURATION has been studied also in the field of theoretical computer science. The problem is PSPACE-complete [9, 19, 41], and there are some tractable cases [6, 12, 19, 20, 26]. A standard reduction from list colorings to independent sets in graphs (e.g., see [33]) gives the following reduction also for reconfiguration problems.

Theorem 3. As an instance of List Coloring RECONFIGURATION, let \( G' \) be a graph with \( N \) vertices and \( M \) edges, and let \( |C| = k \). Then, there is a corresponding instance of Token Jumping such that the input size is \( O(k^2N + kM) \) and its distance is equal to that of the original instance.
Proof. We construct a graph $G$ for Token Jumping, as follows: For each vertex $v$ in $G'$, we construct a new clique of $|L(v)|$ vertices; each vertex in the clique corresponds to a color in $L(v)$. For each edge $uv$ in $G'$, if $L(u) \cap L(v) \neq \emptyset$, then we join the two vertices in the cliques for $u$ and $v$ that correspond to each color $c \in L(u) \cap L(v)$. Then, the theorem follows from the fact that a one-to-one correspondence holds between list colorings of $G'$ and maximum independent sets of $G$.

Bonsma and Cereceda [9, Theorem 21] constructed an instance series for List Coloring Reconfiguration such that the distances of the instances are exponential in the input size. Then, by taking such List Coloring Reconfiguration instances, Theorem 3 gives a hard instance series for Token Jumping, which we call the LC series in this paper.

3.2 Original instance series

In this subsection, we construct our original hard instance series for Token Jumping, which we call the IS series in this paper. We give the following theorem.

Theorem 4. Let $x \geq 3$, $y \geq 1$, $z \geq 1$ be any three integers such that $x$ is odd. Then, there is an instance of Token Jumping such that

- the number of vertices is $5xy + 2z$,
- the number of edges is $8xy - y + 2z - 1$, and
- its distance is $\Omega(z \cdot x^y)$.

More specifically, its distance is equal to

$$\frac{x(x-1)^y-2}{x-2}z + \frac{x(x-1)^y-(2x-4)y-x(3x-2)+2y}{(x-2)^2}.$$ 

Proof sketch. We here sketch the construction of our instance, and roughly explain why it requires an exponential distance. The graph $G$ of the instance is shown in Figure 3. It consists of a path of $2z$ vertices (shown in the left of the figure) and $y$ villages (shown in Figure 4), where each village has $x$ houses (shown in Figure 5). We order the houses from top to bottom, and the villages from left to right, as labeled in Figure 3.

We say that a house is in the north-position (resp., south-position) if the two tokens in the house are placed as shown in Figure 5(a) (resp., in Figure 5(b)). For an independent set $I$ of $G$ and each $i \in \{1, 2, \ldots, y\}$, we denote by $s_i(I)$ the number of houses in the south-position in the $i$-th village; thus $s_i(I) \in \{0, 1, \ldots, x\}$. In the initial independent set $I_s$, all the houses in all the villages are in the north-position, and hence $s_i(I_s) = 0$ for all $i \in \{1, 2, \ldots, y\}$. In the target independent set $I_t$, all the houses in the $y$-th village are in the south-position, and the others are in the north-position; that is, $s_y(I_t) = x$ and $s_i(I_t) = 0$ for all $i \in \{1, 2, \ldots, y-1\}$. Therefore, any reconfiguration sequence from $I_s$ to $I_t$ must change the value $s_y$ for the $y$-th village from 0 to $x$. The structure of $G$ forces the following properties:

- to increase the value $s_y$ from $j$ to $j+1$ for even $j$, we need to change the value $s_{y-1}$ for the $(y-1)$-st village from 0 to $x$; and
- to increase the value $s_y$ from $j$ to $j+1$ for odd $j$, we need to change the value $s_{y-1}$ for the $(y-1)$-st village from $x$ to 0.

During the changes of $s_y$ from 0 to $x$, the above changes for $s_{y-1}$ happen $x$ times alternatively. Furthermore, $y$ villages are recursively connected in $G$, and this recursive structure yields the exponential factor $\Omega(x^y)$ into the distance of our instance. □

\[\text{IS stands for Independent Sets, since this instance series is designed originally for Token Jumping.}\]
Figure 3 The construction of an instance in the IS series, where $x$ is odd.

Figure 4 The village gadgets used in the IS series. (a) The number of houses in the south-position in the village is 0. (b) The number of houses in the south-position in the village is $x$.

Figure 5 The house gadgets used in the IS series. (a) A north-position. (b) A south-position.

4 Solvers Used for Evaluation

In this section, we describe three solvers that will be used for the evaluations in Section 5: SAT-based solver, ZDD-based solver, and IDA-BFS-based solver. The authors develop the SAT-based solver for this paper to perform the most fundamental evaluation, which will be available as described in Conclusions. The ZDD-based and IDA-BFS-based solvers have participated in CoRe Challenge 2022 [39]. The ZDD-based solver solved the instance with the largest shortest reconfiguration sequence in the challenge. The IDA-BFS-based solver solved the most instances and got the first place in the shortest track of the overall solver category. Both competition solvers are publicly available from the competition repository.

7 https://github.com/core-challenge/2022solver-showcase
4.1 SAT-based solver

Let \((G, I_s, I_t)\) be an instance of Token Jumping, where \(G = (V, E)\). Our SAT-based solver employs the method of the bounded model checking [4]. More specifically, for a bound (integer) \(\ell\), the solver determines whether there is a reconfiguration sequence of length \(\ell\) between \(I_s\) and \(I_t\); and then we increment the bound \(\ell\).

First, for an integer \(i \in \{0, 1, \ldots, \ell\}\) and a vertex \(u \in V\), let \(p_{i,u}\) be a propositional variable such that it is true if and only if a token is placed on the vertex \(u\) at Step \(i\); we define the initial independent set \(I_s\) as Step 0. To represent an independent set at Step \(i\), we introduce a function \(\text{Token}(i)\) as follows:

\[
\text{Token}(i) = \begin{cases} 
\bigwedge_{u \in I_s} p_{i,u} \land \bigwedge_{u \notin I_s} \neg p_{i,u} & \text{if } i = 0, \\
\bigwedge_{(u,v) \in E} (\neg p_{i,u} \lor \neg p_{i,v}) & \text{if } 0 < i < \ell, \\
\bigwedge_{u \in I_t} p_{i,u} \land \bigwedge_{u \notin I_t} \neg p_{i,u} & \text{if } i = \ell.
\end{cases}
\]

Next, we introduce two kinds of propositional variables \(q_{i,u}^{10}\) and \(q_{i,u}^{01}\) to represent a token movement at a vertex \(u \in V\) between Steps \(i\) and \(i + 1\): \(q_{i,u}^{10}\) (resp., \(q_{i,u}^{01}\)) is true if and only if a token is removed from \(u\) (resp., placed to \(u\)) between Steps \(i\) and \(i + 1\). Then, the following constraint \(\text{Jump}(i)\) represents a single token movement between Steps \(i\) and \(i + 1\):

\[
\text{Jump}(i) = \left( \bigwedge_{u \in V} (q_{i,u}^{10} \leftrightarrow (p_{i,u} \land \neg p_{i+1,u})) \right) \\
\land \left( \bigwedge_{u \in V} (q_{i,u}^{01} \leftrightarrow (\neg p_{i,u} \land p_{i+1,u})) \right) \\
\land \left( \sum_{u \in V} q_{i,u}^{10} = 1 \right) \land \left( \sum_{u \in V} q_{i,u}^{01} = 1 \right).
\]

Note that the arithmetic constraint of \(\sum_i x_i = 1\) can be encoded into propositional clauses by using the sequential counter [38].

Finally, we check whether there is a reconfiguration sequence of length \(\ell\) by computing the satisfiability of the following formula \(\Psi_\ell\):

\[
\Psi_\ell = \bigwedge_{i=0}^{\ell} \text{Token}(i) \land \bigwedge_{i=0}^{\ell-1} \text{Jump}(i)
\]

Our SAT-based solver increases \(\ell\) one by one and performs satisfiability testing of \(\Psi_\ell\). The method stops once the formula \(\Psi_\ell\) becomes satisfiable, and outputs its model as a shortest reconfiguration sequence. In addition, incremental SAT solving [14] is used to speed up the computation by reusing learned clauses of SAT solvers. We implemented the proposed SAT-based method in the Scala language and used CaDiCaL [5] as its backend SAT solver.

4.2 ZDD-based solver

A zero-suppressed binary decision diagram (ZDD) is a data structure that efficiently represents the family of sets. Ito et al. [23] developed a solver that solves various reconfiguration problems (including Token Jumping) using ZDDs. In CoRe Challenge 2022 [39], the ZDD-based solver solved Token Jumping instances requiring longer shortest reconfiguration sequences that the other participants could not solve.
We briefly describe the algorithm of the ZDD-based solver for Token Jumping. The solver conducts the breadth-first search in the solution space $S_k(G)$ starting from the node corresponding to $I_s$. (Recall the definition of $S_k(G)$ given in Section 2.) Let $Z^i$ be the family of all independent sets in $S_k(G)$ that are distance at $i$ from $I_s$. We represent $Z^i$ as a ZDD in a compressed form and construct $Z^0, Z^1, \ldots$ in turn, where $Z^0 = \{I_s\}$. After constructing $Z^i$, we check whether $Z^i$ contains the target independent set $I_t$. If so, we can conclude that there is a reconfiguration sequence of length $i$ between $I_s$ and $I_t$. Ito et al. [23] designed an algorithm that efficiently constructs $Z^i$ from $Z^{i-1}$ using ZDD manipulation methods.

4.3 IDA-BFS-based solver

Turau and Weyer developed a solver for Token Jumping, which got the first place in the shortest track of the overall solver category in CoRe Challenge 2022 [39]. Their solver is a hybrid one carrying both the iteratively deepening $A^*$ algorithm (IDA*) [29] and the breadth-first search (BFS) on the solution space $S_k(G)$. In this paper, we call their solver the IDA-BFS-based solver.

Roughly speaking, the IDA-BFS-based solver regards Token Jumping as the standard graph problem that searches a path in a graph $S_k(G)$ connecting two nodes $I_s$ and $I_t$. Thus, we can make use of $A^*$ search. However, $A^*$ search needs a huge amount of memory because $S_k(G)$ is usually large and it needs to store the visited nodes. To save the memory usage, the IDA-BFS-based solver employs the method that combines the depth-first search (DFS) with $A^*$ search, called IDA* [29]. In addition, when the answer to $(G, I_s, I_t)$ is no, IDA* searches nodes in $S_k(G)$ many times. To avoid such a situation, the IDA-BFS-based solver uses BFS together with $A^*$, which run in parallel in separate threads. Details are given in the first section of the description of CoRe Challenge 2022 [40].

5 Empirical Evaluations

In this section, we evaluate the instance series proposed in Section 3 by the solvers in Section 4. In the following, all experiments were conducted on a machine equipped with a 3.2 GHz CPU and 64GB of memory. The time limit is 30 minutes for each instance.

5.1 Benchmark instances

We generated benchmark instances from the series of SAT (Section 3.1.1), SP (Section 3.1.2), LC (Section 3.1.3), and IS (Section 3.2). In addition, for the purpose of comparisons, we generated random instances as follows:

1. Generate a random graph by specifying the numbers of vertices and edges. These numbers were chosen to be approximately the same as those from benchmark instances of SAT, SP, LC, and IS for the purpose of comparison.
2. For the generated graph, compute two maximal independent sets of the same size as initial and target independent sets.
3. Check if there exists a reconfiguration sequence for the generated instance. If it exists, return it as a random instance; otherwise, attempt another pair of maximal independent sets. If a given number (this time 100) of attempts is unsuccessful, return to Step 1.

---

8 Specifically, the `gnm_random_graph` method in the NetworkX graph library [18] is used, which chooses a graph uniformly at random from all the graphs with $n$ vertices and $m$ edges for specified $n$ and $m$. 
For each $n$ and $m$, we generated five random instances. In the following, all values for random instances are their median values. For each series, instances with 130 vertices or fewer were generated. As we will describe in Conclusions, all benchmark instances and their generators are available online.

### 5.2 Evaluation

#### 5.2.1 Distances of benchmark instances

First, we evaluated the benchmark instances by their distances, which are computed by one of the three solvers in Section 4. Figure 6 shows the distances, where the horizontal axis represents the number of vertices, and the vertical axis in a logarithmic scale represents the distance.

Recall that we have theoretically shown in Section 3 that distances of SAT, SP, LC and IS series increase exponentially with the numbers of vertices. This fact was also confirmed by our experiments as shown in Figure 6 (notice that the vertical axis uses a logarithmic scale): the plots of SP and IS draw steeply straight lines, and the plots of SAT and LC draw comparably mild straight lines.

On the other hand, the line of RAND no longer increases significantly when the number of nodes is over 100. This implies the difficulty of constructing scalable hard instances for Token Jumping in a straightforward way.

#### 5.2.2 Evaluation by CPU time

Next, we evaluated the benchmark instances by CPU times of three solvers in Section 4. Recall that the SAT-based solver (Section 4.1) was developed by the authors for this paper to perform the most fundamental evaluation. On the other hand, the ZDD-based and IDA-BFS-based solvers (Sections 4.2 and 4.3, respectively) have participated in CoRe Challenge 2022 [39], and they are designed for Token Jumping. Figures 7(a), 7(b), and 7(c) show the CPU times of the SAT-based, ZDD-based, and IDA-BFS-based solvers, respectively. In each figure, the horizontal axis represents the number of vertices, and the vertical axis in a logarithmic scale represents the CPU time.

For the SAT-based solver (see Figure 7(a)), the CPU time increases sharply according to the increase in the number of vertices for all hard instance series SAT, SP, LC, and IS. In particular, hard instances having more than 50 vertices become unsolvable for the
SAT-based solver. In contrast, all instances in RAND are solved within 10 seconds, even when the number of vertices increases to more than 100. We remark that the reason why the computation time of the SAT solver exceeds one second for all instances is believed to be due to the use of the Scala programming language on the Java Virtual Machine for implementation, which results in significant overhead during startup.

The ZDD-based and IDA-BFS-based solvers show similar behaviors, as shown in Figures 7(b) and 7(c), respectively. The plots of CPU time for all hard instance series SAT, SP, LC, and IS draw straight lines, namely showing the exponential growths of CPU times. Compared to the SAT-based solver, lines of SAT, SP, LC, and IS are much more gradual, and indeed the ZDD-based and IDA-BFS-based solvers can solve hard instances having more than 100 vertices. In this sense, our hard instance series are “nicely” hard to develop and improve a solver for Token Jumping.

We note that the lines of RAND for the ZDD-based and IDA-BFS-based solvers are different from that for the SAT-based solver: the ZDD-based and IDA-BFS-based solvers experience rapid increases in CPU times according to the numbers of vertices in those instances. We discuss the reason for this behavior in the following section, regarding the SAT-based and ZDD-based solvers.

5.2.3 Analyses of solver performance

To understand the behavior of solvers, we add some more analyses for the SAT-based and ZDD-based solvers.

We analyzed the number of clauses in the SAT formula for the SAT-based solver because this is a standard metric for the difficulty of SAT instances. Figure 8(a) shows the growth of the number of clauses according to the number of vertices. As this figure shows, the number of clauses exponentially grows for all hard instance series SAT, SP, LC, and IS; while it
becomes stable for RAND even when the number of vertices is over 100. These behaviors are almost the same as the increases in the distances of instances, as shown in Figure 6. Recall that our SAT-based solver employs the bounded model checking which depends on distances of instances, as explained in Section 4.1.

For the ZDD-based solver, we analyzed the number of independent sets that appear in the computation. Recall that $Z^i$ is the family of all independent sets in the solution space $S_k(G)$ that are distance at $i$ from $I_s$, and that the ZDD-based solver constructs $Z^i$ from $Z^{i-1}$ using ZDD manipulation methods. Thus, as a metric, we computed the number of independent sets in $Z^i$, and take the maximum among $i = 0, 1, \ldots, \ell$, where $\ell$ is the distance of the instance. Figure 8(b) shows the growth of this maximum number of independent sets computed by the ZDD-based solver, according to the number of vertices. Then, compared to the line of RAND, those of SAT, SP, LC, and IS are much more gradual, and hence they are exponentially small numbers. This seems to be a natural property, because there are at most $2^{|V|}$ independent sets in a graph $G = (V,E)$, and hence the number of independent sets at distance $i$ would be small if the distance $\ell$ is exponential. Because the ZDD-based solver conducts the breadth-first search on $S_k(G)$, it becomes more effective for hard instances and less effective for random instances in RAND.

5.2.4 Evaluation summary

We summarize the evaluations. As theoretically shown in Section 3, all the proposed instance series SAT, SP, LC, and IS result in exponential growths in the distances according to the number of vertices. On the other hand, as shown in Figure 6, we could not observe such growth in the distance for random instances in RAND. In addition, as mentioned in Introduction, recall that instances in CoRe Challenge 2022 [39] made from graphs in the DIMACS challenge do not have long distances: despite the wide range of vertex numbers from 11 to 10000, 80% of such instances have distance less than 10, and the longest distance is only 112. Therefore, we confirmed the effectiveness of theoretical approaches to creating benchmark instances, particularly when we require scalability. In this context, there is a related work [10] that analyzes the asymptotic behavior of longest distances of instances among all graphs having prescribed numbers of vertices and tokens, although their intention is not to focus on the construction of hard instances.

In addition, we analyzed how this distance property affects the performances of solvers. We confirmed that the required CPU times by three solvers also show scalability and exponential growth according to the number of vertices. This implies that solvers adept at handling long distances demonstrate good performance, resulting in identical rankings for each solver.
Conclusions

In this paper, we proposed four types of instance series for Token Jumping, to accelerate practical studies of general-purpose solvers that do not rely on prior problem-specific knowledge. Our instance series is scalable in the sense that instance scales are controlled by the numbers of vertices of input graphs. We focused on the distances of instances to establish the hardness, and theoretically show that our instance series require exponential numbers of steps with respect to the number of vertices. We confirmed their scalability and hardness by three distinct solvers: SAT-based solver, ZDD-based solver, and IDA-BFS-based solver. We emphasize again that constructing such scalable and hard instances is not a trivial task. In addition, our experiments demonstrated that the randomly generated instances can be sufficiently hard for two types of solvers, but not for the SAT-based solver. All benchmark instances, experimental result logs, and solver programs are available on a GitHub repository.

As a future work, there would be other ways to establish the hardness of instances for Token Jumping. For example, there may be instances where the distance is short but listing all candidate independent sets for the next step is difficult. The difficulty of such instances may depend on other parameters such as the size of independent sets. Future work also includes creating benchmark datasets for other reconfiguration problems.

References


Improved Cut Strategy for Tensor Network Contraction Orders

Christoph Staudt  Friedrich Schiller University Jena, Germany
Mark Blacher  Friedrich Schiller University Jena, Germany
Julien Klaus  Friedrich Schiller University Jena, Germany
Farin Lippmann  Friedrich Schiller University Jena, Germany
Joachim Giesen  Friedrich Schiller University Jena, Germany

Abstract
In the field of quantum computing, simulating quantum systems on classical computers is crucial. Tensor networks are fundamental in simulating quantum systems. A tensor network is a collection of tensors, that need to be contracted into a result tensor. Tensor contraction is a generalization of matrix multiplication to higher order tensors. The contractions can be performed in different orders, and the order has a significant impact on the number of floating point operations (flops) needed to get the result tensor. It is known that finding an optimal contraction order is NP-hard. The current state-of-the-art approach for finding efficient contraction orders is to combine graph partitioning with a greedy strategy. Although heavily used in practice, the current approach ignores so-called free indices, chooses node weights without regarding previous computations, and requires numerous hyperparameters that need to be tuned at runtime. In this paper, we address these shortcomings by developing a novel graph cut strategy. The proposed modifications yield contraction orders that significantly reduce the number of flops in the tensor contractions compared to the current state of the art. Moreover, by removing the need for hyperparameter tuning at runtime, our approach converges to an efficient solution faster, which reduces the required optimization time by at least an order of magnitude.

2012 ACM Subject Classification Theory of computation → Algorithm design techniques; Mathematics of computing → Solvers; Applied computing → Physics

Keywords and phrases tensor network, contraction order, graph partitioning, quantum simulation

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.27


Funding Supported by the Carl Zeiss Stiftung within the project “Interactive Inference”.

1 Introduction
Quantum computing is a rapidly evolving field that holds a lot of promise, especially for simulating quantum systems. However, building reliably working quantum computers is still a major challenge, because of to the fragile nature of quantum states, which can easily be disturbed by environmental factors [3]. Therefore, simulating quantum systems on classical computers is still a viable alternative. It is, however, a computationally intensive task, as the size of the state space grows exponentially with the number of qubits.
Tensor networks are a fundamental data structure used in quantum simulations [16], including the simulation of quantum circuits that are designed to run on quantum computers, but can also be simulated on classical computers. A tensor network is a graph, where the nodes are tensors and the edges represent indices shared among the tensors. Evaluating a tensor network means contracting the tensors over the shared indices, one index after the other. We provide a formal definition of tensor networks and tensor network contractions in Section 2. The efficiency of evaluating a tensor network heavily depends on the order of the indices that is used for the contractions. Finding a good, or even optimal, contraction order is known as the tensor network contraction order problem. Finding an optimal order is NP-hard [12]. Therefore, heuristics are used in practice for computing good contraction orders. There are two main classes of contraction order heuristics, namely greedy [24] and graph-structure-based heuristics, such as tree decompositions, graph partitioning, or community detection [8, 24]. In practice, the best results are reported when combining graph-structure and greedy algorithms [8]. Graph-structure-based algorithms are used to hierarchically decompose the problem into smaller and balanced subproblems and subproblems below a certain size are optimized by a greedy approach [8, 10]. Here, we present a new hybrid of graph-structure and greedy heuristics that addresses three main shortcomings of the state of the art: (1) Relating balanced partitions to balanced contraction trees, (2) disregarding free indices in the cut size, and (3) a weak correlation between partition size and contraction cost.

Experimental results show that our hybrid approach outperforms the state of the art in terms of contraction order quality and optimization time. Compared to the state of the art, our hybrid approach reduces the number of floating point operations (flops) in the tensor contraction on the Google Sycamore circuits [2] by at least a third, in a fraction of the time.

Furthermore, we present an anytime algorithm that does not rely on hard-to-tune hyperparameters. Both aspects are practically relevant as it happens in practice that more time is spent on optimizing contraction orders, including hyperparameter tuning, than on actually contracting the tensor network. An anytime algorithm for contraction order optimization that tracks the required number of flops can stop when it no longer pays off in terms of the overall running time. This includes optimizing the contraction order and actually contracting the tensor network. Experimental results show that our hybrid algorithm exhibits a better contraction order quality to optimization time ratio than state-of-the-art algorithms for contraction order optimization.

2 Tensor Network Contractions

Tensor networks are given by tensors with a shared set of indices. For \( n \in \mathbb{N}^+ \), the set of tensor indices is given as \( [n] = \{1, \ldots, n\} \), the length of the \( i \)-th index is given by \( k_i \in \mathbb{N}^+ \), and the set of positions is the Cartesian product \( K = \prod_{i=1}^{n} [k_i] \). For an index set \( I \subseteq [n] \), a tensor over the real numbers is a mapping

\[
T : K_{|I|} \to \mathbb{R}
\]

that maps positions to real numbers. Here, \( K_{|I|} \) is the projection of \( K \) onto the indices in \( I \). The size \(|I|\) of the index set \( I \) is called the order of \( T \). If \( I = \emptyset \), then \( K_{|\emptyset|} = \{()\} \) contains only the empty tuple and the corresponding order-0 tensor is a scalar. A tensor network is a set \( \{T_1, \ldots, T_m\} \) of tensors that, for all \( i \in [n] \), satisfy

\[
J_i := |\{j \in [m] | i \in I_j\}| \leq 2,
\]

that is, any index is shared by at most two tensors. Indices of axes that belong to only one tensor are called free indices. The structure of a tensor network can be encoded in a weighted graph with a vertex for every tensor and an edge for every shared index. The edge weights
are the lengths of the corresponding axes. We illustrate the graphical representation on the simple example of a matrix chain multiplication that we will use throughout this article. Here, free indices are depicted by dangling edges. Let \( M_1 \in \mathbb{R}^{2 \times 8} \) and \( M_2, M_3, M_4 \in \mathbb{R}^{8 \times 8} \) be four matrices, where \( M_i \) and \( M_j \) share an index if \( |i - j| = 1 \). The corresponding tensor network is shown in Figure 1.

![Figure 1 Matrix chain tensor network. The edges correspond to tensor axes and are labeled by indices. Their weights (in brackets) are the lengths of the axes. Dangling edges belong to free indices.](image)

Let \( L = \{ i \in [n] \mid |J_i| = 2 \} =: \{ i_1 < \ldots < i_\ell \} \) be the set of contraction indices. The tensor network contraction problem asks to compute

\[
\sum_{\alpha \in K_I} \prod_{j=1}^{m} T_j(\alpha_{|_{I_j}}),
\]

where \( \alpha_{|_{I_j}} \) is the projection of \( \alpha \) onto the indices in \( I \). The tensor network contraction problem can also be written in the form

\[
\sum_{\alpha_1 \in [k_1]} \ldots \sum_{\alpha_\ell \in [k_\ell]} \prod_{j=1}^{m} T_j(\alpha_{|_{I_j}}),
\]

where \( \alpha = (\alpha_1, \ldots, \alpha_\ell) \). The interpretation of the latter form is that the edges of the tensor network are contracted one after the other in the order given by the order on \([\ell]\). Changing the contraction order by a permutation \( \pi \) on \([\ell]\) does not change the value. This can be significantly more efficient with a good choice for \( \pi \), because, by the distributive law, products and sums can be interleaved.

In our matrix chain multiplication example, the contraction indices are \( L = \{ j, k, i \} \) with \( k_j = k_k = k_l = 8 \), while the free indices are \( i \) and \( m \) with \( k_i = 2, k_m = 8 \). The tensor network contraction problem for the matrix chain is to compute

\[
R(i, m) = M_1 M_2 M_3 M_4 = \sum_{j=1}^{8} \sum_{k=1}^{8} \sum_{l=1}^{8} M_1(i, j) \cdot M_2(j, k) \cdot M_3(k, l) \cdot M_4(l, m).
\]

By the associativity of matrix multiplication, we have different contraction orders. For instance, \( (M_1 M_2) M_3 M_4 \) corresponds to the order \( jkl \), and

\[
(M_1 M_2)(M_3 M_4) = \sum_{k=1}^{8} \left[ \sum_{j=1}^{8} M_1(i, j) \cdot M_2(j, k) \cdot \sum_{l=1}^{8} M_3(k, l) \cdot M_4(l, m) \right],
\]

where products and sums have been interleaved by the distributive law, corresponds to the orders \( jlk \) and \( ljk \). The contraction orders differ in the number of multiplications (flops), which are 384 for the contraction order \( jkl \) and 768 for the contraction orders \( jlk \) and \( ljk \). Every contraction order can be encoded in a contraction tree, where the tree is uniquely defined by the order, but the tree does not uniquely determine the order. However, all contraction orders of the same tree require the same number of flops. Therefore, it is sufficient...
to find a good contraction tree and not a specific contraction order. The contraction trees for \(((M_1 M_2) M_3) M_4\) and \((M_1 M_2)(M_3 M_4)\) with intermediate tensors, labeled by \(C_1\) and \(C_2\), are shown in Figure 2.

(a) Imbalanced contraction tree for the order \(jkl\). (b) Balanced contraction tree for \(jlk\) and \(ljk\).

Figure 2 Comparison of contraction trees for the matrix chain problem from Figure 1, using different contraction orders.

3 Related Work

Finding an optimal contraction order, that is, a contraction order that minimizes the number of flops, is known to be NP-hard [12]. Therefore, randomized search heuristics [22] and, especially, greedy heuristics [8, 24] are in widespread use. Greedy heuristics score the edges of the tensor network, for instance by the size difference of a contracted tensor and the two tensors that have been contracted. For the contraction ordering, an edge with minimal score is selected, and the scores of the remaining edges have to be updated, because the contraction along the selected edge can affect the scores of the remaining edges. Greedy heuristics build a contraction tree from the bottom up.

Alternatively, contraction trees can also be built top down. Since every subtree of a contraction tree corresponds to a subnetwork of the original tensor network, any contraction tree corresponds to a hierarchical partitioning of the tensor network. Graph partitioning was first used by Kourtis et al. [11] for computing contraction orders. It has become a standard technique for challenging quantum supremacy circuits [9, 17, 18] and is an integral part of \textsc{cotengra}, the state-of-the-art implementation for computing contraction orders [8]. It was integrated into a recently presented quantum circuit simulator targeting the current state of quantum computing, also known as the noisy intermediate-scale quantum (NISQ) era [26]. For a subdivision, \textsc{cotengra} aims at minimizing the cost of the cut, measured by the sum of the edge weights in the cut. The subdivision scheme shares the disadvantage of optimizing locally and not considering contraction costs down the line with the greedy approach. To counteract expensive contractions down the line, \textsc{cotengra} balances the cuts. For computing a balanced, low cost hierarchical partitioning of a tensor network, \textsc{cotengra} builds on the Karlsruhe (hyper-)graph partitioner (\textsc{kahypar}) [23] that constitutes the state of the art. However, \textsc{cotengra} still faces some challenges. Most notably, balanced contraction trees can be suboptimal. A first example can be seen in Figure 2, where the number of flops for the balanced contraction tree is significantly larger than for the unbalanced tree. The problem can be mitigated by increasing the number of partitions and the imbalance parameter of the graph partitioner. Therefore, given a user defined time limit, \textsc{cotengra} searches over these hyperparameter settings. Moreover, \textsc{kahypar} is randomized and also needs several runs. Together, this results in a fairly large variance in the quality of \textsc{cotengra}'s solutions. The issues become evident on a larger example [15], which we also discuss in our experiments in Subsection 5.5.
Motivated by the special case of matrix chain multiplication, where the contraction order problem can be solved by dynamic programming (DP) in polynomial time \cite{4}, Ibrahim et al. first bring the tensors of the tensor network into an optimized linear order and then use the DP approach for matrix chains on the linear order \cite{10}. We included this DP approach in our experimental evaluation. Dynamic programming has also been used by Pfeifer et al. \cite{20} to exhaustively search the space of all contraction orders to find an optimal contraction order. The time and space requirements for this DP approach have been improved by Liang et al. \cite{13}. Still, computing an optimal contraction order is only feasible for small tensor networks.

On the theoretical side, for tensor networks where all indices have the same length, Markov and Shi \cite{14} have shown that the most expensive contraction requires exponentially many flops only in the treewidth of the line graph of the tensor network. Furthermore, given a tree decomposition of the line graph, they have devised an algorithm for computing a contraction order that requires exponentially many flops in the width of the given tree decomposition. The cost of the most expensive contraction can be a reasonable proxy for the overall magnitude of the number of flops if there are not many large contractions. In general, however, it is only a rough approximation for the number of flops \cite{5}. Moreover, the analysis and algorithm apply only to networks without free indices and where all indices have the same length.

Another line of research is to learn good contraction orders. Meirom et al. \cite{15} present a reinforcement Learning (RL) approach combined with graph neural networks (GNNs) for computing contraction orders. While the reported results compare well to the state of the art established by \textit{cotengra}, their performance depends on the training data. At the moment it is difficult to assess the out-of-sample performance of the RL approach in terms of time and quality. Nevertheless, we have included it in our experiments in Section 5.

\section{Algorithmic Improvements}

Top-down algorithms like \textit{cotengra} that build on cost optimized balanced cuts of the tensor network face three main challenges:

1. **Relating balanced partitions to balanced contraction trees.** For many tensor networks the optimal contraction tree is very imbalanced. Current approaches can only generate imbalanced contraction trees, if the partitions are imbalanced. However, highly imbalanced partitions lead to a more greedy-like approach that can produce high contraction costs down the line.

2. **Accounting for free indices in the cut size.** The cut size should relate to the contraction cost between the partitions. Currently, however, free indices, which also contribute to the contraction cost, are discarded before the problem is passed to the graph partitioner. \textit{cotengra} can address this problem by putting all nodes with a free index into a single partition. However, this constraint limits the possible cuts.

3. **Accounting for a node’s contribution to the contraction cost of a subnetwork.** The size of the partitions should relate to contraction cost of the corresponding subnetworks. Current node weighting schemes, however, do not reflect the contributions of individual nodes to the contraction costs very well.

Moreover, by addressing the three challenges, many of the hyperparameters used by \textit{cotengra} can be eliminated. In the following, we present our solutions to the three challenges.
4.1 Arbitrarily balanced contraction trees from balanced partitions

To address the first challenge, that is, relating balanced partitions to balanced contraction trees, we propose a new problem formulation that allows for arbitrarily balanced contraction trees from balanced partitions. Our algorithmic changes can be illustrated on the matrix chain multiplication example from Figure 1. The edge weights passed to the partitioner are the binary logarithms of the index lengths, because the contraction cost between two tensors is proportional to the product of their index lengths, which is equal to the sum of the edge weights, that is, the cut size, after applying the binary-logarithm transformation. Here, we color the free indices gray, because in the standard approach, the graph partitioner receives the network without them. A balanced cut leads to the child and parent networks that are depicted in Figure 3a and Figure 3b, respectively.

![Figure 3](image1)

(a) The two child networks after the cut.

(b) Parent network, combining the two partitions.

Figure 3 The standard approach creates a parent network, which has a child node for each partition. The children can be partitioned further. The edge weights are logarithms of the index lengths. Gray edges represent free indices that are not passed to the graph partitioning algorithm.

Since the initial partition is balanced and there is only one way to contract the parent network, this inevitably leads to the more expensive (balanced) contraction tree in Figure 2b. The problem can be circumvented by picking one of the partitions and declaring it as the parent partition. We will explain later how the parent is picked. Instead of creating a new parent network with two nodes, one for each child, we just combine the nodes of the parent partition with a single child node that represents the result of the other partition as shown in Figure 4. The parent network $P$ can still lead to the balanced tree in Figure 2b if needed by contracting the index $l$ next. However, it can also lead to the imbalanced tree in Figure 2a by contracting the index $k$ next. Thereby we have decoupled balancing the partitions from balancing the contraction tree. This allows us to always run the graph partitioner with a small imbalance parameter, typically resulting in smaller total contraction costs.

![Figure 4](image2)

Figure 4 The child network $C$ (on the left) is contracted first and the resulting tensor is added as node $C$ (yellow) to the new parent network $P$ (on the right). Note that the index $k$ is a free index of the child network $C$.

If the tensor network has free indices, the parent network will be chosen as described in the next section. Otherwise, we compute a contraction order for the two possible parent and child network combinations using a greedy algorithm and pick the one with the lowest total cost.
4.2 Accounting for free indices in the cut size

The second challenge is to account for free indices when computing the cut size, because, without the free indices, the cut size does not accurately represent the contraction cost. It is important to consider free indices even if the original tensor network had none, because every cut introduces free indices into the child networks.

We illustrate the problem on a square tensor network with two free indices that is shown in Figure 5. The minimal balanced cut contains only the edges $a$ and $c$, which suggests a contraction cost of $2 \cdot 2^3 = 16$ for the parent network. However, the child networks $C_1$ and $C_2$ have free indices that contribute to the overall contraction cost. Contracting the parent network and contracting each child network all have the same cost, namely $2 \cdot 2^3 \cdot 2^3 \cdot 2^3 = 1024$ flops, leading to total contraction costs of $3072$ flops.

As shown in Figure 5c, the approach of turning one child network into a parent network and integrating the other child network as a node into the parent network, which we have proposed in Subsection 4.1 to counteract problems with balanced cuts, does not resolve the problem. However, the idea to add additional nodes to the network can also be used to account for indices in the cut size. Here, we add a new node $F$, called free node, to the network and connect it to the existing nodes by the free indices. As can be seen in Figure 6a, a minimal balanced cut in our example network now contains the edges $b$ and $f$, because all other cuts contain at least three edges or produce a partition with only one node. The new node $F$ represents the result tensor of the contraction and thus becomes the root of the contraction tree. Therefore, as depicted in Figure 6b, the child network $C_2$, which contains the node $F$, becomes the parent network $P_2$. The child network $C_1$ is added as a node to $P_2$. The minimal balanced cut now accounts for all edges shared between the two partitions but also for the free indices of the child network. The total contraction cost is the sum of the contraction cost of the child network $C_1$ and the contraction cost of the parent network $P_2$. Contracting $C_1$ requires $2^3 \cdot 2^3 \cdot 2^3 = 512$ flops, and the two remaining contractions in $P_2$ require 1024 flops each. Thus, the total contraction cost is 2560 flops, which is about 30% less than the contraction cost for the standard approach.

As shown in Figure 6c, the approach of turning one child network into a parent network and integrating the other child network as a node into the parent network, which we have proposed in Subsection 4.1 to counteract problems with balanced cuts, does not resolve the problem. However, the idea to add additional nodes to the network can also be used to account for indices in the cut size. Here, we add a new node $F$, called free node, to the network and connect it to the existing nodes by the free indices. As can be seen in Figure 6a, a minimal balanced cut in our example network now contains the edges $b$ and $f$, because all other cuts contain at least three edges or produce a partition with only one node. The new node $F$ represents the result tensor of the contraction and thus becomes the root of the contraction tree. Therefore, as depicted in Figure 6b, the child network $C_2$, which contains the node $F$, becomes the parent network $P_2$. The child network $C_1$ is added as a node to $P_2$. The minimal balanced cut now accounts for all edges shared between the two partitions but also for the free indices of the child network. The total contraction cost is the sum of the contraction cost of the child network $C_1$ and the contraction cost of the parent network $P_2$. Contracting $C_1$ requires $2^3 \cdot 2^3 \cdot 2^3 = 512$ flops, and the two remaining contractions in $P_2$ require 1024 flops each. Thus, the total contraction cost is 2560 flops, which is about 30% less than the contraction cost for the standard approach.

As shown in Figure 5c, the approach of turning one child network into a parent network and integrating the other child network as a node into the parent network, which we have proposed in Subsection 4.1 to counteract problems with balanced cuts, does not resolve the problem. However, the idea to add additional nodes to the network can also be used to account for indices in the cut size. Here, we add a new node $F$, called free node, to the network and connect it to the existing nodes by the free indices. As can be seen in Figure 6a, a minimal balanced cut in our example network now contains the edges $b$ and $f$, because all other cuts contain at least three edges or produce a partition with only one node. The new node $F$ represents the result tensor of the contraction and thus becomes the root of the contraction tree. Therefore, as depicted in Figure 6b, the child network $C_2$, which contains the node $F$, becomes the parent network $P_2$. The child network $C_1$ is added as a node to $P_2$. The minimal balanced cut now accounts for all edges shared between the two partitions but also for the free indices of the child network. The total contraction cost is the sum of the contraction cost of the child network $C_1$ and the contraction cost of the parent network $P_2$. Contracting $C_1$ requires $2^3 \cdot 2^3 \cdot 2^3 = 512$ flops, and the two remaining contractions in $P_2$ require 1024 flops each. Thus, the total contraction cost is 2560 flops, which is about 30% less than the contraction cost for the standard approach.
Improved Cut Strategy for Tensor Network Contraction Orders

4.3 Contraction cost based node weights

The third challenge is to find good estimates for the contraction costs for the partitions. We address this challenge by assigning weights to the nodes of the network, so that the weight of a partition reflects its expected contraction costs.

At the moment, there are two standard node weight functions, namely assigning unit weight to all nodes or to assign the binary logarithm of the node size. We refer to the first weight function as unit weight and the second as log-size weight. Before we introduce our own weight function, we discuss problems with the two standard weight functions. Figure 7a,b shows an example graph with weighted nodes. In the unit weighted case, the smallest balanced cut contains the edges b,c,f, and g, which separates the lower triangle from the other nodes. To use this cut to find a good contraction order, we need to decide which partition becomes the parent and which becomes the child subnetwork. We can then use the opt_einsum greedy algorithm to compute contraction orders on the subnetworks. Regardless which partition is chosen as parent, the total cost is 790 flops. A corresponding contraction tree is shown in Figure 8a. In the log-size weighted case, the smallest balanced cut contains the edges b,c,e,i, and g. Here, the best result is obtained by choosing the two nodes in the middle of the network as the parent subnetwork. However, with 838 flops the cost is even higher than in the unit weighted case. A corresponding contraction tree is shown in Figure 8b. Both weight functions miss that the top middle tensor is rather expensive to contract and needs to be separated from the tensor with indices b,d and e (white) and the tensor with indices e and f (dark yellow). To address this problem, we define a new weight function.

Figure 7 Different node weight functions for the same tensor network give rise to different minimal balanced cuts (depicted by dashed edges).

Figure 8 Contraction trees for the example graph using different node weight functions. The numbers represent the binary logarithms of the contraction costs.

The core idea is to improve on the log-size weight function. For a given contraction tree, the log-size does not necessarily reflect the actual contraction costs that should be attributed to the tensor. For any node of the contraction tree, we can compute the cost of contracting the two associated and already contracted subnetworks. We want to distribute this cost among the tensors in the subnetworks. Intuitively, tensors that contribute many indices to
the node should be assigned more of the cost. This leads to the following idea for a weight function: Given a contraction tree $G$ and a node $T$ of the tensor network, the nodes of $G$ correspond to subnetworks of the tensor network. The set of subnetworks that contains the tensor corresponding to $T$ forms a path from a leaf to the root of $G$. The contraction costs along this path should be partially attributed to $T$ depending on how many indices $T$ shares with the nodes on the path. To keep the weight function simple, we assign the contraction cost times the number of contributed indices to $T$. Formally, let $A$ be a node on the path, let $\text{shared-indices}(T, A)$ be the number of indices shared between $T$ and $A$, and $\text{cost}(A, G)$ be the contraction cost at node $A$ in $G$. The weight assigned to $T$ at $A$ would be

$$\log_2(\text{cost}(A, G)) \cdot \text{shared-indices}(T, A).$$

To mitigate the effect of long paths, we only assign the maximum weight along the path, that is,

$$\text{cost-weight}(T, G) = \max_{A \in \text{path}(T, G)} \left[ \log_2(\text{cost}(A, G)) \cdot \text{shared-indices}(T, A) \right].$$

Coming back to our example, the greedy contraction order gives rise to the contraction tree in Figure 8a. Using this contraction tree results in the cost weight function that is shown in the weighted network in Figure 7c. The smallest balanced cut in this weighted network contains the edges $b, d, i$, and $f$. Here, the best result is obtained by choosing the two lower right nodes of the network as the parent subnetwork. The resulting total contraction costs, computed as before, are only 290 flops. A corresponding contraction tree is shown in Figure 8c.

## 5 Experiments

We will first describe the datasets and baselines that we use in our experiments, then we quickly go over implementation details of our algorithm. Afterwards, we discuss the impact of our algorithmic modifications on the contraction cost for the two largest networks in the dataset. Finally, we present comparative experimental results on all networks. Our code, including a python implementation of the algorithm and the experiments, is available at https://github.com/ti2-group/hybrid_contraction_tree_optimizer.

The experiments were run on a computer with two Intel Xeon 6140 processors with 18 Cores, 2.3 Ghz clock speed and 192 GB RAM.

### 5.1 Datasets and baselines

Unfortunately, we were not able to run the code from Meirom et al. [15] for lack of a suitable GPU cluster that is needed for training, and for missing pre-trained weights for inference. Nevertheless, since Meirom et al. published all networks that they have used in their experiments, we benchmark us and all other baselines against them. For their approach, we report the numbers as given in their paper. We clarified with the authors that they used the opt_einsum reporting for flops. In contrast to our description so far, opt_einsum counts additions and multiplications leading to twice as many flops compared to counting only multiplications. To be consistent, we report the flops counted by opt_einsum in all experiments and all algorithms.

Meirom et al. [15] have used the following three types of tensor networks in their experiments:

1. **Synthetic random 3-regular networks**, which were generated with the Python package opt_einsum [24]. The number of tensors ranges from 25 to 225 and for each network size there are 100 different networks. We excluded networks with only 25 tensors.
from our benchmarks, because they can be solved by an optimal dynamic programming algorithm in less than a second. The number of indices per tensor was sampled i.i.d. from \{2, 3, 4, 5, 6\}.

2. The Sycamore circuits from the Google quantum supremacy experiment [2]. Meirom et al. have used the circuits with 53 qubits and \( m = 10, 12, 14 \) and 20 cycles. These networks have 162 to 379 tensors after simplification with the Python package quimb [6].

3. A single Max-cut circuit with 1,688 qubits and depth 21 resulting in 5,908 tensors with four free indices. Meirom et al. modeled the network based on previous work [19]. Since the network is much larger than the others used in their experiments, they only optimized the last 100, 1,000, and 2,000 contraction steps. To that end, they computed a contraction order using the \texttt{opt_einsum} greedy heuristic and then contracted the network up to the step they wanted to optimize. In their published data, the file for the last 2,000 steps was missing, so we generated one ourselves and made sure that our implementation matches theirs for the last 100 and 1,000 steps. For the last 100 steps, the overall cost is dominated by the previous steps, so we will not report the results for that. The costs for contracting up to the last 1,000 and 2,000 steps are 1.61e+12 and 4.49e+08, respectively. In order to always compare the costs for the entire network, we added these preprocessing flops to the results.

Besides Meirom et al.'s reinforcement learning based approach, named RL-TNCO from now on, we benchmark the following baseline algorithms that also have been used in [15]:

- The greedy and kahypar algorithms from the Python package cotengra [8]. We will refer to them as CTG greedy and CTG kahypar in the following. Because the greedy algorithm now utilizes an updated Rust implementation and because faster implementations allow for more trials, the results have improved compared to the earlier benchmarks [15].
- cotengra offers three hyperparameter optimizers: \texttt{random}, \texttt{nevergrad} and \texttt{optuna} [21, 1]. For the greedy algorithm, mostly the \texttt{random} hyperparameter optimizer works best. It is fast and thus allows for more trials. Only on large networks with long run times, \texttt{optuna} can provide some benefit. For the kahypar algorithm, \texttt{optuna} generally seems to give superior results compared to \texttt{nevergrad} and \texttt{random}. Additionally, \texttt{cotengra} offers various methods to improve a given path, which are commonly used in practice. However, since this is possible for paths produced by any algorithm, we excluded this feature to simplify our comparison.
- The minimal linear arrangement and dynamic programming approach, called MLA-DP [10]. We integrated the Julia implementation of MLA-DP into cotengra. Unfortunately, MLA-DP’s hyperparameters are not documented. Changing the hyperparameters renders the algorithm unstable. Therefore, we only vary the seed randomly. Moreover, MLA-DP does neither support networks with more than one connected component nor networks with free indices. Since the random regular networks include disconnected networks and the max-cut network has free indices we cannot benchmark MLA-DP on these networks.

5.2 Implementation details

We first describe small further improvements of the algorithm and then detail the hyperparameters that are used by our algorithm and its dependencies.

- To decide which part of the network should be partitioned next, the algorithm uses a priority queue. The priority queue is sorted by the contraction costs of the subnetworks, which are calculated by a greedy algorithm. While the priority queue is not empty,
the subnetwork with smallest contraction cost is removed from the priority queue and partitioned into a parent and a child network. Partial contraction orders are computed for both networks, again by the greedy algorithm. The partition is rejected if the sum of the costs of the two partial contraction orders is larger than the cost of the old contraction order. Otherwise, both subnetworks are added to the priority queue with the cost of the respective partial contraction order.

- Since networks commonly have more than one minimal balanced cut of the same size, we repeat each partitioning step several times and choose the partition with the lowest total amount of flops. The number of repetitions is a hyperparameter that we describe below.
- As described in Subsection 4.3, the weights that we assign to the nodes depend on a contraction tree. To make the weights less dependent on a single contraction tree and guiding the partitioning towards a more balanced cut, we average the weights over the contraction trees that result from the repetitions that we have described in the item above.
- To turn our algorithm into an anytime algorithm, we use the total costs of the greedy contraction trees to update our current cost estimate for the whole network.
- Since both cotengrust [7] and kahypar also support (tensor) hypernetworks, our algorithm does so as well.

Our algorithm is using the following hyperparameters and default choices for them:

- **Partition imbalance parameter**: We configure kahypar to partition the networks into two parts with an imbalance parameter of 0.05.
- **Choice of greedy algorithm**: We use the greedy algorithm implemented in cotengrust. We run the greedy algorithm 256 times with random choices for its hyperparameters.
- **Node weight function**: We use only cost based node weights, as they give superior results in our ablation study in Subsection 5.3.
- **Number of repetitions of the partitioner on a given (sub) network**: 10.
- **Base case**: If a subnetwork has less than 15 nodes we compute an optimal contraction tree by the optimal dynamic programing algorithm by Pfeifer et al. [20] that is implemented in cotengrust.
- **Stopping criterion**: The algorithm stops if the cost of the next subnetwork scheduled for partitioning, and therefore the cost for each remaining subnetwork, is less than 0.001% of the current contraction cost.

The exact versions of the used softwares in our implementation can be found in the code repository.\(^1\)

### 5.3 Ablation study

Figure 9 shows how our algorithmic ideas reduce the number of flops required to contract the max-cut and Sycamore network for \(m = 20\), where \(m\) is the number of cycles. On the x-axis we compare how different mappings of the tensor network to the graph partitioning problem influence the number of flops. Since the max-cut network has output indices adding a free node proves to be effective. It reduces the required number of flops by a fifth. Without the free node, all the other algorithmic modifications are not effective. This is because, without the free node, as shown in Figure 10, the algorithm cannot find a cut that improves the greedy baseline. Using the cost weighted nodes improves the result slightly.

---

\(^1\) [https://github.com/ti2-group/hybrid_contraction_tree_optimizer/blob/main/requirements.txt](https://github.com/ti2-group/hybrid_contraction_tree_optimizer/blob/main/requirements.txt)
Improved Cut Strategy for Tensor Network Contraction Orders

The results on the Sycamore network are very different. Here, every algorithmic modification improves the result. However, the overall improvement is smaller. Again, the improvements are a result of more successful cuts as can be seen in Figure 10. While the Sycamore network does not have free indices, new free indices are introduced into the child network of each cut. Therefore, the free node proves to be effective in this setting as well. Additionally, weighting the nodes based on the contraction tree results in a clear improvement.

**Figure 9** Comparison of the number of flops for the standard partitioning approach, the integrated child node, and the integrated child and free node modifications. The colors encode how the nodes are weighted. Each hyperparameter combination is executed ten times.

**Figure 10** Comparison of the number of cuts for the different algorithmic modifications that result in a reduction of contraction costs compared to the cost estimation by the greedy algorithm.

5.4 Contraction order quality, run time, and variance

Here, we present comparative results on the random regular networks and the Sycamore circuits. We compare the contraction order quality, the runtimes, and the variance of the results. Results for the max-cut network will be presented in the next subsection.
5.4.1 Synthetic random regular networks

RL-TNCO was only benchmarked on networks up to size 100. They trained a network for each size on randomly generated graphs of the same size and then evaluated them on the benchmark set. For each problem they sampled 50 and 1000 paths and reported the results for both separately. For all methods the experiment was run three times per network and then averaged over all networks of the same size [15]. The median was first calculated for each network and then over all networks of the same size. All results are shown in Table 1 and Table 2. On small networks with only 50 tensors the reinforcement learning approach outperforms the other methods. However, on larger networks it falls behind, at 100 nodes it performs worse than all other algorithms. Our algorithm achieves the best mean and median cost on all networks with more than 50 tensors. On the larger networks our algorithm finds up to 40 times better paths on average for a runtime of 300s. For shorter runtimes the difference becomes even larger. In general our algorithm improves only slightly when the run time is increased from one minute to 5 minutes, whereas cotengra improves significantly. This might indicate that cotengra would need more time to find good paths. However, as we will see in the next section, even running CTG kahypar for hours may not result in much better paths.

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Number of flops on the synthetic networks with 50 to 125 nodes. The run times in parenthesis belong to RL-TNCO.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean ± std</td>
</tr>
<tr>
<td>n</td>
<td>50</td>
</tr>
<tr>
<td>Scale</td>
<td>10^7</td>
</tr>
<tr>
<td>run time</td>
<td>5s (4.2s)</td>
</tr>
<tr>
<td>CTG greedy</td>
<td>10.2 ± 41.7</td>
</tr>
<tr>
<td>CTG kahypar</td>
<td>8.29 ± 22.9</td>
</tr>
<tr>
<td>RL-TNCO</td>
<td>3.03 ± 4.54</td>
</tr>
<tr>
<td>this paper</td>
<td>5.67 ± 13.1</td>
</tr>
<tr>
<td>run time</td>
<td>60s (74.6s)</td>
</tr>
<tr>
<td>CTG greedy</td>
<td>6.02 ± 30.4</td>
</tr>
<tr>
<td>CTG kahypar</td>
<td>5.55 ± 13.8</td>
</tr>
<tr>
<td>RL-TNCO</td>
<td>3.22 ± 3.53</td>
</tr>
<tr>
<td>this paper</td>
<td>5.25 ± 12.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2</th>
<th>Number of flops on the synthetic networks with 150 to 225 nodes.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean ± std</td>
</tr>
<tr>
<td>n</td>
<td>150</td>
</tr>
<tr>
<td>run time</td>
<td>60s</td>
</tr>
<tr>
<td>CTG greedy</td>
<td>33.8 ± 154</td>
</tr>
<tr>
<td>CTG kahypar</td>
<td>10.6 ± 49.6</td>
</tr>
<tr>
<td>this paper</td>
<td>0.43 ± 0.05</td>
</tr>
<tr>
<td>run time</td>
<td>300s</td>
</tr>
<tr>
<td>CTG greedy</td>
<td>14.6 ± 56.3</td>
</tr>
<tr>
<td>CTG kahypar</td>
<td>1.43 ± 34.4</td>
</tr>
<tr>
<td>this paper</td>
<td>0.34 ± 0.84</td>
</tr>
</tbody>
</table>
5.4.2 Sycamore networks

To compare against RL-TNCO, we first present results on the Sycamore networks for an optimization time of 1 hour. We report the results after 3 hours for RL-TNCO, since they did not report their results after 1h for the smaller networks. As shown in Table 3 we outperform all other approaches on all networks. This even holds when our algorithm runs for only one minute. We present extended statistics in Table 4 in the appendix.

![Flops vs. time.](image1.png)  
(a) Flops vs. time.  

![Variance (1h).](image2.png)  
(b) Variance (1h).

Figure 11 Comparison of flops over time and variance after 1h for the Sycamore network with \( m = 20 \).

Besides contraction order quality measured in flops, practical relevant performance measures are quality over run time and variance of the solution quality. Therefore, we present the flops over time and the variance after 1h for the \( m = 20 \) network in Figure 11. We run each algorithm 10 times for 10, 30, 60, 120, 360, 600, 1800 and 3600 seconds. Our algorithm reliably finds good contraction trees after 360s, while the other algorithms slowly improve towards the 1h time limit. However, as can be seen in Figure 11b, even after 1h the quality of their results, especially of CTG kahypar varies greatly.

Table 3 Median number of flops on the Sycamore networks over 10 independent runs. The parameter \( m \) represents the cycles of the Sycamore circuits and controls their size and complexity.

<table>
<thead>
<tr>
<th></th>
<th>( m = 10 )</th>
<th>( m = 12 )</th>
<th>( m = 14 )</th>
<th>( m = 20 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scale</td>
<td>( 10^{10} )</td>
<td>( 10^{12} )</td>
<td>( 10^{14} )</td>
<td>( 10^{18} )</td>
</tr>
<tr>
<td>MLA-DP 1h</td>
<td>2.45</td>
<td>10.7</td>
<td>1.94</td>
<td>8.32</td>
</tr>
<tr>
<td>CTG greedy 1h</td>
<td>2.34</td>
<td>8.61</td>
<td>2.21</td>
<td>4.51</td>
</tr>
<tr>
<td>CTG kahypar 1h</td>
<td>2.20</td>
<td>10.9</td>
<td>2.06</td>
<td>6.87</td>
</tr>
<tr>
<td>RL-TNCO 3h</td>
<td>5.44</td>
<td>7.4</td>
<td>2.63</td>
<td>3.5</td>
</tr>
<tr>
<td>this paper 60s</td>
<td>1.78</td>
<td>6.05</td>
<td>1.61</td>
<td>2.79</td>
</tr>
<tr>
<td>this paper 1h</td>
<td><strong>1.52</strong></td>
<td><strong>5.76</strong></td>
<td><strong>1.37</strong></td>
<td><strong>2.46</strong></td>
</tr>
</tbody>
</table>
5.5 Hyperparameter optimization and max-cut network

So far, all reported experimental results for our algorithm were obtained with the default hyperparameter settings. Since we observed that the hyperparameter optimizer optuna for CTG greedy is able to find better contraction orders than CTG greedy with randomly chosen hyperparameters on the large max-cut network, we also examined the effect of optimizing the imbalance hyperparameter on our algorithm. Results are shown in Figure 12 and in the appendix in Table 5. On the max-cut network, both cotengra algorithms struggle to find a good contraction order. CTG kahypar finds a reasonably good contraction order, but only after 6h. The same holds for RL-TNCO, which finds a good contraction order after nearly 6h. Even though the network is large, the contraction costs are rather low. On most modern computers it can be contracted within minutes. For the long optimization times to pay off, one needs to contract the network fairly often. Our algorithm with default hyperparameter settings finds a good contraction order after only 10 minutes. Optimizing the imbalance parameter improves the results by about 25%, leading to the best contraction order for this network.

![Flops vs. time.](image1)

![Variance (1h).](image2)

**Figure 12** Comparison of flops over time and variance after 1h for the last 2000 steps of the max-cut network.

6 Conclusions

We have introduced a novel cut strategy for finding efficient tensor network contraction orders. The strategy consists of three algorithmic modifications over the state-of-the-art approach based on top-down balanced graph partitioning and bottom-up greedy algorithms.

Our first modification addresses a mismatch between the partition balance objective and imbalanced optimal contraction trees. For many tensor networks the optimal contraction tree is imbalanced, which is difficult to achieve with a balanced cut objective in the partitioner. In our modification, we designate one partition as the child and the other as the parent network. The child network is integrated as a node into the parent network, which facilitates the generation of both unbalanced and balanced contraction trees. Moreover, the modification reduces the number of hyperparameters.
Our second modification addresses that free indices were not accounted for in the cut size objective. Accounting for free indices is important, because they contribute to the contraction cost. In our modification, we include an additional node that represents the free indices in each partitioning step. The additional node ensures that the contraction costs are more accurately reflected in the cut objective.

Our third modification addresses the problem of estimating the contraction costs of partitions. A partitioner can use the estimates as weights when computing a small balanced cut. In our modification we address this challenge by assigning weights to the nodes of the network, so that the weight of a partition reflects its expected contraction costs. To iteratively update the weights we leverage the greedy approach for computing contraction orders on the partitions. The iterative refinement ensures that the weighted network graph more accurately reflects the optimization objective.

Experimental results demonstrate that all three algorithmic modifications can improve the current state of the art in finding good contraction orders. Overall, the proposed algorithm computes more efficient contraction orders in significantly less time. Even though our algorithm is randomized, just like the baseline algorithms, it exhibits a reduced variance in the quality of the computed solutions. Moreover, the algorithm serves as an anytime algorithm that can stop when it becomes obvious that further optimizations no longer pay off.

References


C. Staudt, M. Blacher, J. Klaus, F. Lippmann, and J. Giesen


A Comparison of hyperoptimizers

In the following Figure 13 we compare different hyperoptimizers on the max-cut and Sycamore \((m = 20)\) networks. While randomly choosing hyperparameters for the CTG greedy algorithm outperforms the other hyperoptimizers on the Sycamore network, optimizing the hyperparameters with optuna pays off for the Max-Cut network. Optimizing the hyperparameters for CTG kahypar always leads to better contraction orders. Overall optuna is the better choice in this case as well. Note that on the Max-Cut network we only report a runtime of 6h since, smaller run times lead to too much variance in the results.

![Figure 13](image-url) Comparison of hyperoptimizers on the max-cut and Sycamore \((m = 20)\) networks.
### Table 4 Number of flops for the sycamore networks aggregated over 10 independent runs.

<table>
<thead>
<tr>
<th>m</th>
<th>10</th>
<th>12</th>
<th>14</th>
<th>20</th>
<th>10</th>
<th>12</th>
<th>14</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scale</td>
<td>10⁻⁰</td>
<td>10⁻¹</td>
<td>10⁻²</td>
<td>10⁻³</td>
<td>10⁻⁰</td>
<td>10⁻¹</td>
<td>10⁻²</td>
<td>10⁻³</td>
</tr>
<tr>
<td>MLA-DP 1min</td>
<td>2.80 ± 0.13</td>
<td>12.0 ± 0.2</td>
<td>1.15 ± 0.14</td>
<td>9.59 ± 0.45</td>
<td>2.78 ± 0.09</td>
<td>12.3 ± 0.33</td>
<td>2.11 ± 0.05</td>
<td>9.67 ± 0.33</td>
</tr>
<tr>
<td>CTG greedy 1min</td>
<td>3.24 ± 0.31</td>
<td>13.4 ± 2.43</td>
<td>2.40 ± 0.45</td>
<td>5.34 ± 0.55</td>
<td>3.30 ± 0.16</td>
<td>13.5 ± 0.99</td>
<td>2.71 ± 0.28</td>
<td>5.40 ± 0.52</td>
</tr>
<tr>
<td>CTG kahypar 1min</td>
<td>2.47 ± 0.43</td>
<td>31.3 ± 6.97</td>
<td>4.09 ± 1.63</td>
<td>8.31 ± 0.64</td>
<td>2.46 ± 0.36</td>
<td>31.7 ± 10.33</td>
<td>3.61 ± 0.85</td>
<td>8.51 ± 0.25</td>
</tr>
<tr>
<td>thin paper 1min</td>
<td>1.79 ± 0.10</td>
<td>6.05 ± 0.15</td>
<td>1.59 ± 0.06</td>
<td>2.76 ± 0.16</td>
<td>1.76 ± 0.05</td>
<td>6.05 ± 0.03</td>
<td>1.61 ± 0.04</td>
<td>2.79 ± 0.09</td>
</tr>
<tr>
<td>MLA-DP 10min</td>
<td>2.59 ± 0.08</td>
<td>10.0 ± 0.46</td>
<td>2.00 ± 0.05</td>
<td>8.85 ± 0.52</td>
<td>2.60 ± 0.03</td>
<td>10.7 ± 0.31</td>
<td>2.00 ± 0.04</td>
<td>8.95 ± 0.38</td>
</tr>
<tr>
<td>CTG greedy 10min</td>
<td>2.78 ± 0.26</td>
<td>9.67 ± 1.45</td>
<td>2.41 ± 0.18</td>
<td>4.87 ± 0.36</td>
<td>2.80 ± 0.21</td>
<td>9.41 ± 0.90</td>
<td>2.09 ± 0.12</td>
<td>4.86 ± 0.16</td>
</tr>
<tr>
<td>CTG kahypar 10min</td>
<td>2.34 ± 0.29</td>
<td>16.0 ± 2.87</td>
<td>2.19 ± 0.15</td>
<td>7.34 ± 0.26</td>
<td>2.34 ± 0.27</td>
<td>16.0 ± 1.12</td>
<td>2.14 ± 0.06</td>
<td>7.49 ± 0.20</td>
</tr>
<tr>
<td>thin paper 10min</td>
<td>1.60 ± 0.09</td>
<td>5.79 ± 0.16</td>
<td>1.42 ± 0.04</td>
<td>2.57 ± 0.11</td>
<td>1.58 ± 0.06</td>
<td>5.76 ± 0.14</td>
<td>1.43 ± 0.02</td>
<td>2.59 ± 0.06</td>
</tr>
<tr>
<td>MLA-DP 1h</td>
<td>2.47 ± 0.04</td>
<td>10.7 ± 0.36</td>
<td>1.93 ± 0.03</td>
<td>8.24 ± 0.58</td>
<td>2.45 ± 0.00</td>
<td>10.7 ± 0.36</td>
<td>1.94 ± 0.03</td>
<td>8.32 ± 0.25</td>
</tr>
<tr>
<td>CTG greedy 1h</td>
<td>2.35 ± 0.23</td>
<td>8.58 ± 0.87</td>
<td>2.20 ± 0.07</td>
<td>4.48 ± 0.17</td>
<td>2.34 ± 0.16</td>
<td>8.61 ± 0.03</td>
<td>2.21 ± 0.04</td>
<td>4.51 ± 0.09</td>
</tr>
<tr>
<td>CTG kahypar 1h</td>
<td>2.25 ± 0.33</td>
<td>11.1 ± 1.16</td>
<td>2.14 ± 0.29</td>
<td>6.51 ± 0.82</td>
<td>2.20 ± 0.27</td>
<td>10.9 ± 0.26</td>
<td>2.06 ± 0.09</td>
<td>6.87 ± 0.37</td>
</tr>
<tr>
<td>RL-TNCO 1h</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>thin paper 1h</td>
<td>1.50 ± 0.04</td>
<td>5.73 ± 0.13</td>
<td>1.34 ± 0.08</td>
<td>2.45 ± 0.07</td>
<td>1.52 ± 0.03</td>
<td>5.76 ± 0.08</td>
<td>1.37 ± 0.03</td>
<td>2.46 ± 0.03</td>
</tr>
<tr>
<td>CTG greedy 3h</td>
<td>2.24 ± 0.21</td>
<td>7.92 ± 0.49</td>
<td>2.18 ± 0.05</td>
<td>4.15 ± 0.27</td>
<td>2.24 ± 0.15</td>
<td>7.77 ± 0.30</td>
<td>2.20 ± 0.04</td>
<td>4.09 ± 0.07</td>
</tr>
<tr>
<td>CTG kahypar 3h</td>
<td>2.21 ± 0.29</td>
<td>11.6 ± 2.20</td>
<td>2.05 ± 0.14</td>
<td>6.00 ± 1.02</td>
<td>2.15 ± 0.25</td>
<td>11.0 ± 1.00</td>
<td>2.01 ± 0.05</td>
<td>5.97 ± 0.93</td>
</tr>
<tr>
<td>RL-TNCO 3h</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>5.44 ± N/A</td>
<td>7.4 ± N/A</td>
<td>2.63 ± N/A</td>
<td>3.5 ± N/A</td>
</tr>
<tr>
<td>thin paper 3h</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>2.41 ± 0.04</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>2.40 ± 0.03</td>
</tr>
</tbody>
</table>

### Table 5 Number of flops for the max cut networks aggregated over 10 independent runs.

<table>
<thead>
<tr>
<th>Last steps</th>
<th>Scale</th>
<th>1000</th>
<th>2000</th>
<th>all</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean ± std</td>
<td>10⁻³</td>
<td>10⁻²</td>
<td>10⁻¹</td>
<td>10⁻⁰</td>
</tr>
<tr>
<td>CTG greedy 10min</td>
<td>133 ± 4.06</td>
<td>176 ± 11.2</td>
<td>55.0 ± 0.53</td>
<td>54.9 ± 0.42</td>
</tr>
<tr>
<td>CTG kahypar 10min</td>
<td>729 ± 192±0±15</td>
<td>1.22±6±2.68±06</td>
<td>648±66±2.05±67</td>
<td>549 ± 0.38</td>
</tr>
<tr>
<td>thin paper 10min</td>
<td>15.4 ± 0.25</td>
<td>10.3 ± 0.37</td>
<td>15.0 ± 4.55</td>
<td>14.1 ± 2.65</td>
</tr>
<tr>
<td>thin paper (optimized): 10min</td>
<td>10.4 ± 0.00</td>
<td>7.49 ± 0.03</td>
<td>9.04 ± 1.22</td>
<td>8.88 ± 0.66</td>
</tr>
<tr>
<td>CTG greedy 30min</td>
<td>130 ± 6.92</td>
<td>168 ± 11.6</td>
<td>54.9 ± 0.55</td>
<td>54.9 ± 0.36</td>
</tr>
<tr>
<td>CTG kahypar 30min</td>
<td>796 ± 143</td>
<td>162 ± 144</td>
<td>4347 ± 796</td>
<td>486 ± 120</td>
</tr>
<tr>
<td>thin paper 30min</td>
<td>15.0 ± 0.36</td>
<td>10.2 ± 0.27</td>
<td>11.5 ± 0.46</td>
<td>10.8 ± 0.25</td>
</tr>
<tr>
<td>thin paper (optimized): 30min</td>
<td>10.4 ± 0.00</td>
<td>7.49 ± 0.01</td>
<td>8.07 ± 0.31</td>
<td>8.00 ± 0.09</td>
</tr>
<tr>
<td>CTG greedy 1h</td>
<td>125 ± 7.00</td>
<td>160 ± 8.85</td>
<td>54.7 ± 0.33</td>
<td>54.8 ± 0.24</td>
</tr>
<tr>
<td>CTG kahypar 1h</td>
<td>68.4 ± 21.7</td>
<td>64.4 ± 26.6</td>
<td>283 ± 526</td>
<td>82.3 ± 38.2</td>
</tr>
<tr>
<td>thin paper 1h</td>
<td>14.9 ± 0.22</td>
<td>10.0 ± 0.32</td>
<td>11.4 ± 0.48</td>
<td>10.8 ± 0.25</td>
</tr>
<tr>
<td>thin paper (optimized): 1h</td>
<td>15.0 ± 0.18</td>
<td>10.2 ± 0.25</td>
<td>11.3 ± 0.25</td>
<td>10.8 ± 0.25</td>
</tr>
<tr>
<td>CTG greedy 3h</td>
<td>123 ± 3.48</td>
<td>149 ± 9.56</td>
<td>54.4 ± 0.54</td>
<td>54.8 ± 0.31</td>
</tr>
<tr>
<td>CTG kahypar 3h</td>
<td>42.3 ± 7.53</td>
<td>33.1 ± 4.74</td>
<td>46.1 ± 14.4</td>
<td>41.2 ± 7.72</td>
</tr>
<tr>
<td>thin paper 3h</td>
<td>N/A</td>
<td>9.61 ± 0.34</td>
<td>10.8 ± 0.41</td>
<td>10.8 ± 0.41</td>
</tr>
<tr>
<td>thin paper (optimized): 3h</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>CTG greedy 6h</td>
<td>N/A</td>
<td>N/A</td>
<td>54.2 ± 0.54</td>
<td>54.1 ± 0.42</td>
</tr>
<tr>
<td>CTG kahypar 6h</td>
<td>N/A</td>
<td>N/A</td>
<td>3.8 ± 11.5</td>
<td>33.7 ± 0.71</td>
</tr>
<tr>
<td>RL-TNCO 5.5</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>9.29 ± N/A</td>
</tr>
</tbody>
</table>
Finding the Minimum Cost Acceptable Element in a Sorted Matrix

Sebastián Urrutia
Molde University College, Norway

Vinicius dos Santos
Federal University of Minas Gerais, Brazil

Abstract
In this work we introduce the problem of finding a minimum cost acceptable element in an \( n \times n \) matrix \( M \) whose columns and rows are sorted in non-decreasing order. More precisely, given a sorted matrix \( M \) and access to a given oracle function \( f : \mathbb{N} \times \mathbb{N} \rightarrow \{\text{True, False}\} \), one has to find a pair \( (i, j) \) of indices such that \( f(i, j) \) returns \text{True} and the value \( M[i, j] \) is as small as possible.

Assuming the computation of \( f(i, j) \) takes time bounded by a constant, a naive algorithm scanning all the positions of the matrix takes time \( O(n^2) \). Another natural approach, based on a priority queue, takes time \( O(z \log z) \) in which \( z \) stands for the position of the first pair of indices for which the oracle returns \text{True} in a sorted list of all elements of \( M \). In the worst case, when \( z = n^2 \), the naive algorithm is better than the priority queue one. In this work we introduce different algorithms with complexities depending on \( n \) and \( z \), such as \( O(n\sqrt{z}) \) and \( O(\min(n^2, z^2)) \), and compare them, both theoretically and experimentally, in terms of running time and number of calls to the oracle.

Among other things, we find that in most cases our algorithms do not make a significantly larger number of calls to the oracle than the priority queue-based algorithm, which achieves the minimum of such call when all elements of the matrix are distinct, while being much faster in large instances.

2012 ACM Subject Classification
Theory of computation → Design and analysis of algorithms

Keywords and phrases
Search, Sorted matrix, Oracle function, Algorithm complexity

Digital Object Identifier
10.4230/LIPIcs.SEA.2024.28

Funding
Vinicius dos Santos: Partially funded by FAPEMIG grant APQ-01707-21 and CNPq grants 312069/2021-9, 406036/2021-7, 404479/2023-5.

1 Introduction
Consider two tasks to be performed simultaneously. Both of them need some resources (time, space, devices, raw material, manpower, etc.) in order to be executed. Tasks may be done in a given number \( n \) of modes and each of their \( n \) modes has an associated cost. Since resources are shared by the two tasks, some pairs of modes of execution (one mode for each task) may be incompatible. Then, it is natural to think on a problem in which one needs to assign an execution mode to each task such that the assigned modes are compatible and the cost of choosing both modes simultaneously is as small as possible.

We assume that the \( n \) execution modes for each task are given sorted in such a way that mode \( (i, j) \), corresponding to the \( i^{th} \) mode for the first task and the \( j^{th} \) mode of the second task, is at least as costly as modes \((i - k, j), 0 \leq k < i\) and \((i, j - k), 0 \leq k < j\). Also, we are given an oracle function \( f(i, j) \) that returns \text{True} if and only if the \( i^{th} \) mode for the first task is compatible with the \( j^{th} \) mode of the second task.

As a real application consider a Berth and Crane Allocation Problem (BCAP) [6] in which two vessels need to use port facilities to deliver a number of containers and pick-up some others. Both vessels need space, time, manpower and equipment to complete their tasks. The amount and quality of the resources assigned to the tasks (execution modes) determine a financial cost that should be paid to the port operator and the resources available at the...
port must be shared by the two vessels. In this case, the vessels owner would like to find a pair of compatible modes (one for each task) with minimum cost in the sorted matrix $M$ that stores the cost of every pair of execution modes.

2 Formalization

Let $M$ be a non-decreasingly sorted matrix of size $n \times n$ and $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$ a boolean function (oracle function) that takes a pair of valid indices of the matrix as parameters. The problem consists in determining a pair of indices $(i, j)$ such that $f(i, j) = \text{True}$ and $M[i, j]$ is as small as possible. If no such pair exists the problem has no solution. To the best of our knowledge, such problem has not been previously studied in the literature.

Function $f$ is treated here as a black box that indicates if a pair of elements, denoted by their indices, is acceptable or not. There is no other way to determine if a pair of indexes is acceptable, we assume no knowledge on the nature of $f$. For our theoretical results we will assume that function $f$ takes $O(1)$ time to compute. Since in practice the time to compute $f$ can be of relevance, in our experimental analysis we will also consider oracle functions with different computational costs.

Let us define $s$ as an array of pairs of indices of $M$ sorted non-decreasingly by $M[i, j]$. Then, we define $z$ as an unknown parameter indicating the position of the first acceptable pair of indexes in $s$, that is, $f(s[z]) = \text{True}$ and $f(s[\ell]) = \text{False}$ for every $1 \leq \ell < z$. If there is no pair $(i, j)$ with $f(i, j) = \text{True}$ we set $z = n^2$. In the case of ties, $z$ may take any value among the positions of the tied pairs. In that case, the value of $z$ is not well defined. This fact may complicate the theoretical analysis of the algorithms performed in Section 4. For that reason, we assume in that section that there are not repeated values in $M$, for simplicity. Observe that the situation can be handled by using the indices $i$ and $j$ for tie breaking, or considering that in case of ties acceptable elements occur after the non-acceptable ones of the same cost.

One special case of sorted matrices are the so-called $X + Y$ matrices [3, 4, 5]. Let $X, Y$ be two vectors of dimension $n$ sorted in non-decreasing order. The matrix $X + Y$ resulting from the Cartesian sum of $X$ and $Y$ is the matrix $A$ in which $A[i, j] = X_i + Y_j$. In many applications, including versions of BCAP [6], the cost associated with a pair of tasks is just the sum of their individual costs, which can be modelled as $X + Y$ matrices.

3 Related works

Sorted matrices have been considered in the literature and many natural problems have been studied, including sorting, selection and search problems.

The complexity of sorting a sorted square matrix into a linear array has not been shown lower than the complexity of sorting a general matrix, that is, the best known algorithms take time $\Omega(n^2 \log(n))$. For the special case of $X + Y$ matrices, [3] gives a non-constructive proof showing that the elements of these matrices can be sorted with just $O(n^2)$ comparisons. In [5] an algorithm with such number of comparisons is presented, nevertheless the time complexity of the algorithm is still $\Theta(n^2 \log(n))$.

The computation of statistics such as the median value or, more generally, obtaining the $i^{th}$ smallest value in a sorted matrix has been studied, and is sometimes called selection problem. Algorithms for this problem have been presented in [2, 7] with time complexity $O(n)$. Simpler algorithms with the same complexity are obtained in [4].
The search problem consists in, given a value \( v \), find a pair of indices \((i,j)\) such that \( M[i,j] = v \). In [1] the authors show that the problem can be solved in \( \Theta(n) \) and that the complexity is optimal.

4 Algorithms

In this section we present six algorithms for the problem introduced in the previous sections. We start from scratch with a naive algorithm and then we exploit the structure of the input in order to find more efficient algorithms. It is important to remark that, since function \( f \) is given as a black box, a time complexity \( \Omega(n^2) \) is unavoidable in some instances. For instance, if there is a unique pair \((i,j)\) for which \( f(i,j) \) is True, any algorithm that ignores \((i,j)\) could not be correct. Also, finding an acceptable pair does not ensure its optimality, implying a lower bound of \( \Omega(z) \). Hence, in many algorithms we will be interested in strategies that could work well in practice, even if their worst case complexity are still as bad as the naive algorithm.

4.1 A naive algorithm

The simplest way to deal with the problem is to evaluate all elements of \( M \) requiring \( \Theta(n^2) \) operations.

One can evaluate \( f(i,j) \) for all the \( n^2 \) pairs of valid indices and return the acceptable pair with the smallest value in \( M \). This algorithm takes \( \Theta(n^2) \) time.

The procedure can be easily modified by not testing any pair of indices dominated by an already known acceptable pair. That is, if \((i,j)\) is known to be acceptable, there is no need to test the acceptability of any pair \((\ell,p)\) such that \( M[\ell,p] \geq M[i,j] \). Depending on how the pairs are scanned (fixing the first index first, fixing the second index first, etc) and the distribution of acceptable pairs, the modification may not improve the computational time of the algorithm and may worsen it. For that reason, this kind of modification will not be further analysed.

4.2 A sort-based algorithm

The second natural approach to the problem would be to sort the elements of \( M \) in order to evaluate them sorted by cost and stopping the evaluation when the first pair of acceptable indices is found. As mentioned before, even using the best known algorithm for sorted matrices, the sorting step would require \( \Theta(n^2 \log n) \) time, which would dominate the time complexity. Note that, after the sorting step, it is possible to stop the algorithm as soon as the first acceptable pair is found.

Instead of sorting the whole input, it is possible to exploit the structure of the input, which is partially sorted, in order to make use of the previous observation. Recall that we denote by \( s \) the array containing pairs of indices sorted non-decreasingly by \( M[i,j] \) and that \( z \) is the position in \( s \) for the first pair \( p \) for which \( f(p) \) is True. By using a priority queue, it is possible to devise an algorithm requiring \( O(z \log z) \) operations.

In the sort-based algorithm, the priority queue will ensure that the pairs are evaluated in order of their value in \( M \), to make it possible to stop the execution as soon as a viable pair is found. The queue starts with the pair \((1,1)\) and every time the pair \((i,j)\) is taken out of the queue for testing its acceptability, the pairs \((i+1,j)\) and \((i,j+1)\) are added to the queue if they are valid indices and were not previously inserted in the queue. Implementing the priority queue with a binary heap, the algorithm runs in \( O(z \log z) \).
Finding the Minimum Cost Acceptable Element in a Sorted Matrix

A technical issue must be addressed in order to be able to check whether a pair was already inserted in the queue without using too much additional space. By breaking ties in the priority queue by the sum of the indices of the pairs, one can be sure that, if the pair \((i, j)\) is in the queue, no pair \((i + t, j + u)\) with both \(t\) and \(u\) positive is in the queue. Therefore, the elements on the queue form the border of an orthogonal convex region. Then, no more than \(2n - 1\) pairs are in the priority queue at any point in the algorithm execution. Also, note that due to this fact, in order to know if a pair \((i, j)\) that is about to be added to the queue is already there, we need only to compare \(j\) with the maximum value \(k\) such that \((i, k)\) is in the queue. In consequence the algorithm can be implemented using no more than \(O(n)\) additional space: \(O(n)\) for the binary heap and \(O(n)\) for storing the maximum second index in the queue for each possible value of the first index.

Observe that the \(O(z \log z)\) running time could be much better than the \(\Theta(n^2)\) time of the naive approach, since \(z\) can be small when compared to \(n^2\) or even compared to \(n\). Indeed, if \(z = O(n^{2-\epsilon})\) for any \(\epsilon > 0\), the sort-based approach would be asymptotically better. However, if \(z = \Theta(n^2)\), the complexity of the algorithm becomes \(O(n^2 \log(n))\) which can be worse than the naive approach.

### 4.3 Exploiting the main diagonal of the matrix

The next two algorithms to be presented here exploit the main diagonal of \(M\) and require \(O(n\sqrt{z})\) operations. Asymptotically, these algorithms dominate the trivial algorithm as their complexity is \(O(n^2)\) even in the worst case when \(z = \Theta(n^2)\) and are better when the value of \(z\) is small.

The algorithms are based on evaluating with function \(f\) up to iteration \(\ell\), at least all the \(z\) pairs of indices \((i, j)\) such that \(M[i, j] \leq M[\ell, \ell]\).

The following simple theorem form the base of the two next algorithms.

**Theorem 1.** Let \(i, j, k \in \{1, \ldots, n\}\) such that \(i \times j \geq k\). Then \(M[i, j] \geq M[s[k]]\)

**Proof.** Since \(M\) is sorted, \(M[i, j] \geq M[i', j']\) for \(1 \leq i' \leq i\) and \(1 \leq j' \leq j\). There are \(i \times j \geq k\) possible distinct values for \(i'\) and \(j'\). Then, the position of the pair \((i, j)\) in \(s\) is at least \(k\).

An immediate consequence of the theorem is that, when our following algorithms finish iteration \(\lceil \sqrt{z} \rceil\) one can be sure that all the first \(z\) elements of \(s\) has been evaluated with function \(f\).

### 4.4 L-shape algorithm

As the name may suggest, L-shape algorithm will proceed by iterating through lines and columns of \(M\) in increasing order. From now on, in all of our algorithms we will assume that algorithms will be implemented with lazy-evaluation, meaning that if there are two or more conditions in a loop of control sequence, they will be evaluated in order and only if all previous conditions hold.

As shown in Algorithm 1 this approach evaluates in iteration \(\ell\) all the not previously evaluated elements of \(M\) in the line \(\ell - 1\) and in column \(\ell - 1\) that are better than the best known acceptable pair, or all elements, if no acceptable element is known so far. By the end of one iteration of the main loop, for an integer \(\ell\) all elements in lines or columns smaller than \(\ell\) will have been visited (or ignored if a better acceptable solution was already found).

Note that the algorithm does not necessarily stop after the iteration in which it first finds an acceptable pair. It stops when the element in the main diagonal is larger than the best (smallest) acceptable pair found. At this point, we know that no better element would be evaluated by continuing the algorithm execution, and the correctness follows.
Let $best$ be smallest cost of an acceptable element and let $\ell$ be the largest integer such that $M[\ell, \ell] < best$. From Theorem 1, $z \geq \ell^2$. Since $bestpair$ have at least one coordinate bounded by $\ell + 1$, the algorithm main loop is executed at most $\ell + 1 = O(\sqrt{z})$ times, with each iteration performing $O(n)$ steps, resulting in a $O(n\sqrt{z})$ running time. Also note that this is an in-place algorithm, that is, it requires just constant extra space.

### 4.5 Contour-line algorithm

As in the previous algorithm the Contour-line algorithm exploits elements in the main diagonal to bound its running time. However, instead of visiting a whole line (column) at once, in iteration $\ell$, only elements that are smaller or equal to $M[\ell, \ell]$ are evaluated. This way, at the end of iteration $\ell$, the elements already evaluated form a contour line in the matrix bounded by the value of $M[\ell, \ell]$. Let $t_\ell$ be the number of elements of $M$ with value at most $M[\ell, \ell]$. Then, by the end of iteration $\ell$ the algorithm has evaluated exactly the first $t_\ell$ elements of $s$. Once an acceptable solution is found, which happens for the smallest $\ell$ for which $t_\ell \geq z$, the algorithm stops. In that regard it is similar to the sort-based algorithm, evaluating elements bounded by certain value at each step, while it avoids the extra computations related to the upkeep of the heap.

Notice that, contrary to the previous algorithm, the Contour-line stops at the end of the iteration in which an acceptable pair of indices is found as no better pair would be evaluated in future iterations. Also, whenever the best acceptable pair is in the main diagonal of $M$, the Contour-line algorithm performs as few evaluations of the oracle function as the sort-based algorithm (assuming no ties), that is, as few as possible.
Algorithm 2 Contour-line Algorithm. Evaluates just pairs with value smaller than or equal to pair \((\ell, \ell)\) up to iteration \(\ell\).

**Data:** \(M, f\)

\[
\text{for } j \leftarrow 1 \text{ to } n \text{ do} \\
\quad \text{contourline}[j] \leftarrow 1 \\
\quad i \leftarrow 1, \text{ best } \leftarrow +\infty \\
\text{while } i \leq n \text{ and best } = +\infty \text{ do} \\
\quad \text{for } j \leftarrow 1 \text{ to } n \text{ do} \\
\quad \quad \text{while contourline}[j] \leq n \text{ and } M[j, \text{contourline}[j]] \leq M[i, i] \text{ do} \\
\quad \quad \quad \text{if } M[j, \text{contourline}[j]] < \text{best} \text{ and } f(j, \text{contourline}[j]) = \text{true} \text{ then} \\
\quad \quad \quad \quad \text{best } \leftarrow M[j, \text{contourline}[j]] \\
\quad \quad \quad \quad \text{bestpair } \leftarrow (j, \text{contourline}[j]) \\
\quad \quad \quad \text{contourline}[j] \leftarrow \text{contourline}[j] + 1 \\
\quad \quad i \leftarrow i + 1 \\
\quad \text{if best } < +\infty \text{ then} \\
\quad \quad \text{return bestpair} \\
\quad \text{else} \\
\quad \quad \text{return “no solution”}
\]

Let \(\ell\) be the number of iterations of the main loop of the algorithm. As in the analysis of Algorithm 1, no more than \(\ell = O(\sqrt{n})\) iterations are needed. The cost of iteration \(i\), is proportional to \(n\) plus the number of times that an element of \(\text{contourline}\) is incremented. But notice that the value of \(\text{contourline}[j]\) cannot exceed \(i\) when \(j > i\), implying that by the end of the algorithm elements of \(\text{contourline}\) were incremented at most \(\ell n + (n - \ell)\ell\) times, resulting in a \(O(n\sqrt{n})\) complexity.

In contrast to L-Shape algorithm, Contour-line requires \(\Theta(n)\) additional space to store the contour line.

### 4.6 Expanding Square Algorithm

The next algorithm consists in two stages.

In the first stage, only elements in the square defined by the pairs \((1, 1)\) and \((k, k)\) are evaluated up to iteration \(k\). This first stage ends after finishing an iteration in which an acceptable pair is found. Let \((i, j)\) be the best pair for which \(f(i, j)\) tested \(\text{true}\) during this first stage. If there is an acceptable pair better than \((i, j)\), it must be of the form \((i - t, j + u)\) or \((i + t, j - u)\) with both \(t\) and \(u\) positive.

In the second stage, just elements outside of the sub-matrix already evaluated that are smaller than the best pair found are evaluated.

Let \(\ell = \max\{i, j\}\) and suppose first that the pair in position \(z\) of vector \(s\) is not in the \(\ell \times \ell\) sub-matrix the algorithm visited. Let \((i_{\text{opt}}, j_{\text{opt}}) = s[z]\) and \(\ell_{\text{opt}} = \max\{i_{\text{opt}}, j_{\text{opt}}\}\).

The number of positions of \(M\) visited during the execution of Algorithm 3 is bounded by \(\ell^2\) in the first stage, plus a maximum of \(2(\ell(\ell_{\text{opt}} - \ell))\) until finding \((i_{\text{opt}}, j_{\text{opt}})\), plus at most \(z\) elements that are smaller than \(M[i_{\text{opt}}, j_{\text{opt}}]\) and are evaluated after \((i_{\text{opt}}, j_{\text{opt}})\) is found.

We have that \(\ell^2 + 2(\ell(\ell_{\text{opt}} - \ell)) + z = \ell^2 + 2\ell_{\text{opt}} - 2\ell^2 + z = 2\ell_{\text{opt}} - \ell^2 + z < 2\ell_{\text{opt}} + z\).

Since \(s[z]\) is not in the \(\ell \times \ell\) submatrix, \(\ell_{\text{opt}} > \ell\). Clearly, we also have \(z \geq \ell_{\text{opt}}\). Then, \(2\ell_{\text{opt}} + z \leq \min\{3\ell^2, 3z^2\}\).
Algorithm 3  Expanding square algorithm.

Data: $M, f$

if $f(1, 1) = \text{true}$ then
    return $(1, 1)$

$i \leftarrow 2$, $\text{best} \leftarrow +\infty$

while $i \leq n$ and $\text{best} = +\infty$ do
    for $j \leftarrow 1$ to $i$ do
        if $M[i, j] < \text{best}$ and $f(i, j) = \text{true}$ then
            $\text{best} \leftarrow M[i, j]$
            $\text{bestpair} \leftarrow (i, j)$

        if $M[j, i] < \text{best}$ and $f(j, i) = \text{true}$ then
            $\text{best} \leftarrow M[j, i]$
            $\text{bestpair} \leftarrow (j, i)$

    $i \leftarrow i + 1$

if $\text{best} = +\infty$ then
    return “no solution”

if $i \leq n$ then
    $\ell \leftarrow i - 1$

    while $i \leq n$ and ($M[i, 1] < \text{best}$ or $M[1, i] < \text{best}$) do
        $j \leftarrow 1$

        while $j < \ell$ and ($M[i, j] < \text{best}$ or $M[j, i] < \text{best}$) do
            if $M[i, j] < \text{best}$ and $f(i, j) = \text{true}$ then
                $\text{best} \leftarrow M[i, j]$
                $\text{bestpair} \leftarrow (i, j)$

            if $M[j, i] < \text{best}$ and $f(j, i) = \text{true}$ then
                $\text{best} \leftarrow M[j, i]$
                $\text{bestpair} \leftarrow (j, i)$

            $j \leftarrow j + 1$

        $i \leftarrow i + 1$

    return $\text{bestpair}$

When the pair $(i, j)$ found in the first stage is the best acceptable pair, the value of $\ell = \max\{i, j\}$ equals the size of the square. Since $z \geq \ell$, the number of elements visited by the algorithm during the first stage is $\ell^2 \leq z^2$. In the second stage only elements smaller than $M[i, j]$ are visited and those are limited by $z$. Then the total number of elements evaluated are limited by $2z^2$.

The algorithm never evaluates a position more than once and then the complexity of this in-place algorithm is $O(\min\{n^2, z^2\})$. Also, the algorithm requires just constant additional space.

4.7 Level curves algorithm

One implication of Theorem 1 is that if the cheapest acceptable pair is in position $(i_{\text{opt}}, j_{\text{opt}})$, then $z \geq i_{\text{opt}} \times j_{\text{opt}}$. The algorithm presented in this subsection explores this idea by splitting elements in sets $S_1, \ldots, S_n$ such that $S_k = \{(i, j) \mid (k - 1)n < ij \leq kn\}$ and evaluating these sets in order. Once an acceptable pair is found, its cost is used to prune the exploration, similarly as in the contour-line algorithm.
Finding the Minimum Cost Acceptable Element in a Sorted Matrix

More precisely, in the $k$-th iteration the algorithm evaluates elements of $S_k$, in increasing order of coordinates. As in previous algorithms, there is a variable $best$ containing the cost of the best acceptable pair found so far. Whenever evaluating a pair $(i, j)$, if $M[i, j] > best$, no further elements in the same line are considered in the current iteration or in further ones. We remark that $\Theta(n)$ information is stored to make this pruning possible, by storing a list of indices $i$ (lines of the matrix $M$) which can still be searched.

Let $(i_{opt}, j_{opt})$ be the optimal pair and let $k_{opt} = \lceil i_{opt}/j_{opt} \rceil$. By construction, the pair $(i_{opt}, j_{opt})$ will be found in iteration $k_{opt}$. Up to that point, exactly $\lceil k_{opt}/n \rceil$ elements with first coordinate $i$ are visited. So, the number of visited pairs is bounded by

$$\sum_{i=1}^{n} \min(\frac{k_{opt} \cdot n}{i}, n) \leq \sum_{i=1}^{n} \frac{k_{opt} \cdot n}{i} = k_{opt} \cdot n \sum_{i=1}^{n} \frac{1}{i} = O(k_{opt} \cdot n \log n).$$

Since $(k_{opt} - 1)n < i_{opt}/j_{opt}$ and $i_{opt}/j_{opt} \leq z$, then the running time up to iteration $k_{opt}$ can be bounded by $O(z \log n)$ when $k_{opt} > 1$ and by $O(n \log n)$ in the first iteration. Putting everything together, we get an upper bound of $O((n + z) \log n)$. In case $k_{opt}$ is very large, this bound is not tight, since the min operation in the inequality becomes more relevant. It is not hard to see that the algorithm never performs more than $O(n^2)$ operations.

Notice, however, that the algorithm has no way to identify that the desired acceptable element has been found. It remains to bound the running time of the rest of the execution.

During further iterations, we still consider sets $S_k$ in order. Whenever visiting a pair $(i, j)$, either it has smaller cost than $(i_{opt}, j_{opt})$, or we update a auxiliary data structure to never consider elements with the same first coordinate. Hence, the total number of steps after finding the solution is bounded by $O(n + z)$.

Putting all together, we have a $O(\min((n + z) \log n), n^2)$-time $\Theta(n)$-space algorithm.

### 4.8 Comparison and summary of the algorithms

With the exception of the sort-based algorithm, none of the algorithms we discussed have asymptotically worse running time than the Naive algorithm, as illustrated in Table 1. As for space, three algorithms can be implemented using only additional constant space. Contour-line, keeps a vector of $n$ positions to track the contour line, the sort-based, that requires a heap or another non-trivial data structure and the level curves algorithm stores a linear array indicating where to stop the evaluation in each line of $M$.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complexity</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>S-based</td>
<td>$O(z \log z)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>L-shape</td>
<td>$O(n \sqrt{z})$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Contour-line</td>
<td>$O(n \sqrt{z})$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Expanding square</td>
<td>$O(\min(n^2, z^2))$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Level curves</td>
<td>$O(\min((n + z) \log n), n^2)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

Many of the times complexities are incomparable. Besides, it is not reasonable to assume the value of $z$ is known beforehand. If some estimation of the value of $z$ is available in a specific setting, one algorithm can be preferable over another. But besides the worst-case analysis, the algorithms present differences and subtleties, some of which are not captured by Table 1, and the number of steps performed for a given instance could be much smaller than the worst case analysis suggests. We discuss a few of them in the rest of this section and other aspects will be brought to light when analysing the experiments, in Section 5.
L-shape and contour-line have the same worst-case time complexity, but different space complexity. But it is not hard to conceive instances for which contour-line would run faster than L-shape, implying that the extra space bring advantages in some cases. On the other hand L-shape could make best use of cache, by visiting long sequences of consecutive elements, in particular in the case of $X + Y$ matrices, stored implicitly.

Expanding square algorithm shares many similarities with contour-line algorithm, in the sense that in both cases there is a value $\ell$ such that all elements “inside the $\ell$-square”, i.e. elements $(i, j)$ with $i, j \leq \ell$, are evaluated and all other elements considered satisfy $\min(i, j) \leq \ell$. Among elements outside the square, while contour-line only considers elements smaller than $M[\ell, \ell]$, Expanding square consider elements smaller than the element found, which is potentially smaller. However, this implicit square could be different in each algorithm: Contour-line is able to find an acceptable element outside the square, aborting the execution of the algorithm with a much smaller value of $\ell$. Finally, as the complexity analysis suggests, Expanding square may be fast when the cheapest acceptable element is close to the main diagonal, since most of the calls to oracle function $f$ were made to pairs of smaller cost than the optimal, and hence unavoidable.

As the previous discussion suggests, the position of the optimal element, in addition to the value of $k$, may impact the running time of the algorithms. Another instance of this can be observed when comparing algorithms L-shape and level curves. It is not hard to convince oneself that, after $k$ iterations of the main loop of both algorithms, the set of positions considered by L-shape is a subset of the ones considered by level curves. Besides, L-Shape is a simpler algorithm and it is likely that in any implementation it would have smaller constants. Consider the optimal element is in position $(c, n - d)$ for small constants $c$ and $d$. Then both algorithms will visit this element around the same iteration and, from the previous discussion, L-Shape would find it faster. On the other hand, if the element is close to the main diagonal, level curves algorithm has a chance to be much faster than L-Shape.

Suppose, for example, that the desired element is in position $(\sqrt{n}, \sqrt{n})$. Level curves would find it in the first iteration and spend at most $O(n + z)$ additional steps before finishing its execution, to a total of $O(n \log n + z)$ steps, while L-Shape run $\sqrt{n}$ iterations, possibly spending $O(n \sqrt{n})$ steps, which could be much bigger if $z = o(n \sqrt{n})$.

5 Experiments

In this section we discuss our computational tests of the algorithms developed in the previous sections. We first describe how we constructed our benchmark instances and the computational environment in which the experiments were run. Then, we show the results from the experiments.

5.1 Instances

For constructing instances we define a set of parameters detailed in Table 2. By combining different values of the parameters we defined a set of 60 problem instances for which the algorithms were tested.

The instances created are denoted by the value of their parameters. The first number corresponds to the size of the matrix, the second and third refer to the pair of indexes reporting the position of the optimal acceptable element, the forth indicates which is the probability of elements with cost larger than the optimal to be acceptable, the fifth defines the matrix construction method and the last the oracle speed. For example, an instance named I.500.300.310.1.0.1 is an instance with a $500 \times 500$ matrix in which the optimal acceptable pair is $(300, 310)$, elements with larger cost have probability of $1/500$ of being acceptable, the matrix is constructed with the 0 method and a slow oracle is used.

5 Experiments

In this section we discuss our computational tests of the algorithms developed in the previous sections. We first describe how we constructed our benchmark instances and the computational environment in which the experiments were run. Then, we show the results from the experiments.

5.1 Instances

For constructing instances we define a set of parameters detailed in Table 2. By combining different values of the parameters we defined a set of 60 problem instances for which the algorithms were tested.

The instances created are denoted by the value of their parameters. The first number corresponds to the size of the matrix, the second and third refer to the pair of indexes reporting the position of the optimal acceptable element, the forth indicates which is the probability of elements with cost larger than the optimal to be acceptable, the fifth defines the matrix construction method and the last the oracle speed. For example, an instance named I.500.300.310.1.0.1 is an instance with a $500 \times 500$ matrix in which the optimal acceptable pair is $(300, 310)$, elements with larger cost have probability of $1/500$ of being acceptable, the matrix is constructed with the 0 method and a slow oracle is used.
Table 2 Parameters for instance construction.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>Positive integer</td>
<td>Matrix $M$ is of size $n \times n$.</td>
</tr>
<tr>
<td>$(i^<em>, j^</em>)$</td>
<td>Pair of valid indices of $M$</td>
<td>Position of the optimal acceptable element.</td>
</tr>
<tr>
<td>$e$</td>
<td>${0, 1, 2}$</td>
<td>Probability $p$ of a pair $(i, j)$ such that $M(i, j) &gt; M(i^<em>, j^</em>)$ being acceptable:</td>
</tr>
<tr>
<td>v</td>
<td>${0, 1, 2, 3}$</td>
<td>Construction of $M$:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: $X + Y$, $X$ and $Y$ with elements increasing by a random amount between 0 and 10 (inclusive) from one element to the next.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: $X + Y$, $X$ and $Y$ with each element being twice the previous one.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2: $X + Y$, $X$ and $Y$ where the difference between consecutive elements at positions $i + 1$ and $i$ is half the difference between elements at positions $i$ and $i - 1$.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3: all elements have the same value.</td>
</tr>
<tr>
<td>$t$</td>
<td>${0, 1}$</td>
<td>Complexity of oracle:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: Fast oracle, returns the answer in $O(1)$ time.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Slow oracle, perform $O(n)$ dummy operations before returning the answer.</td>
</tr>
</tbody>
</table>

We notice that all matrices we use in our experiments are $X + Y$ matrices. This allow us to compute the cost of any position on the matrix as the sum of the two respective position in the vectors. In that way we avoid dealing with very big matrices in memory, which would be a bottleneck in some experiments. Another reason for the choice is that in real problems (in particular in the one that inspired this work) the sorted matrix frequently is a $X + Y$ matrix.

5.2 Computation Environment
All algorithms were implemented in C and compiled with gcc distributed by Apple (Apple clang version 14.0.0, clang-1400.0.29.102) with no compilation flags. The experiments were run on a MacBook Air running a Mac OS 12.7.2, with a 2,2 GHz Intel Core i7 Dual-Core processor and 8GB 1600MHz DDR3 memory.

5.3 Results
In Tables 3, 4 and 5 we show the obtained computational results for the algorithms developed in this work. Tables 3 and 4 report on the number of evaluations of the oracle for all the six algorithms using several instances with $n = 50$ in Table 3 and $n = 500$ in Table 4. Finally, Table 5 shows computational times obtained for all the algorithms except for the naïve one. We show results with $n = 5000$ and a slow oracle and with $n = 20000$ and a fast oracle. Computational time is displayed in seconds. In all the tables, best results are written in bold.

Several interesting facts regarding the number of evaluations of the oracle can be noted from Table 3 and confirmed from Table 4:
Table 3 Number of calls of the oracle function for small instances.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Naive</th>
<th>S-based</th>
<th>L-shape</th>
<th>C.-line</th>
<th>E. Square</th>
<th>Level curves</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.50.25.0.0.0</td>
<td>2500</td>
<td>1288</td>
<td>1825</td>
<td>1296</td>
<td>1309</td>
<td>1461</td>
</tr>
<tr>
<td>I.50.5.0.0.0</td>
<td>2500</td>
<td>45</td>
<td>385</td>
<td>45</td>
<td>48</td>
<td>115</td>
</tr>
<tr>
<td>I.50.50.0.0.0</td>
<td>2500</td>
<td>1323</td>
<td>1330</td>
<td>1314</td>
<td>2459</td>
<td>1330</td>
</tr>
<tr>
<td>I.50.25.1.0.0</td>
<td>2500</td>
<td>1288</td>
<td>1341</td>
<td>1296</td>
<td>1309</td>
<td>1340</td>
</tr>
<tr>
<td>I.50.5.1.0.0</td>
<td>2500</td>
<td>45</td>
<td>188</td>
<td>45</td>
<td>48</td>
<td>112</td>
</tr>
<tr>
<td>I.50.50.1.0.0</td>
<td>2500</td>
<td>1323</td>
<td>1330</td>
<td>1314</td>
<td>1416</td>
<td>1330</td>
</tr>
</tbody>
</table>

1. The Naive algorithm as expected does $n^2$ evaluations of the oracle for all instances.
2. In many of the instances the algorithms L-shape, contour-line and expanding square perform a number of evaluations similar to the sort-based algorithm which makes exactly $z$ evaluations. Level curves algorithm is also competitive, but when the pair $(i^*, j^*)$ have small coordinates, the algorithm makes a number of unnecessary evaluations in its first (unique) iteration.
3. In some cases, the last four algorithms performed fewer calls of the oracle than the sort-based algorithm. This is only possible due to the presence of elements of the matrix with the same value. This happens more often when method 3 (all elements on the matrix have the same value) is used for the construction of the matrix.
Finding the Minimum Cost Acceptable Element in a Sorted Matrix

Table 4: Number of calls of the oracle function for medium-sized instances.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Naive</th>
<th>S-based</th>
<th>L-shape</th>
<th>C-line</th>
<th>E. Square</th>
<th>Level curves</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.50.250.250.0.0.0</td>
<td>250000</td>
<td>120771</td>
<td>187000</td>
<td><strong>120757</strong></td>
<td>120959</td>
<td>148483</td>
</tr>
<tr>
<td>1.500.5.0.0.0</td>
<td>250000</td>
<td><strong>42</strong></td>
<td>3985</td>
<td><strong>42</strong></td>
<td>44</td>
<td>1060</td>
</tr>
<tr>
<td>1.50.5.500.0.0.0</td>
<td>250000</td>
<td>136165</td>
<td>136152</td>
<td><strong>136151</strong></td>
<td>249535</td>
<td>136152</td>
</tr>
<tr>
<td>1.50.55.55.0.0.0</td>
<td>250000</td>
<td><strong>6296</strong></td>
<td>51085</td>
<td>6305</td>
<td>6348</td>
<td>16877</td>
</tr>
<tr>
<td>1.50.250.250.1.0.0</td>
<td>250000</td>
<td>120771</td>
<td>122390</td>
<td>120757</td>
<td>120959</td>
<td>122276</td>
</tr>
<tr>
<td>1.500.5.5.1.0.0</td>
<td>250000</td>
<td><strong>42</strong></td>
<td>1709</td>
<td><strong>42</strong></td>
<td>44</td>
<td>852</td>
</tr>
<tr>
<td>1.50.550.1.0.0</td>
<td>250000</td>
<td>136165</td>
<td>136152</td>
<td><strong>136151</strong></td>
<td>138425</td>
<td>136152</td>
</tr>
<tr>
<td>1.50.55.55.1.0.0</td>
<td>250000</td>
<td><strong>6296</strong></td>
<td>7363</td>
<td>6305</td>
<td>6348</td>
<td>6976</td>
</tr>
</tbody>
</table>

4. The L-shape algorithm gets bad results when the optimal acceptable pair of indices lay in the main diagonal of the matrix and specially if that pair is the only acceptable pair. In that case, until that pair is visited the algorithm evaluates entire lines and columns of the matrix.

5. The expanding square algorithm gets bad results when the optimal acceptable pair of indices lay far away from the main diagonal of the matrix and specially if that pair is the only acceptable pair. In that case, the square in which each element is evaluated has to be expanded to cover almost the whole matrix.

6. Level curves algorithm seems to deal better with the extreme cases of previous items: when simultaneously compared to L-shape and expanding square, it is never worse than both. In particular, in instances for which L-shape performs well among all algorithms, Level curves has a slightly worse performance. However, for instances where L-Shape performed badly, level curves had a much better result.

7. Parameter $e$ controlling if other elements beside the optimal are acceptable only affects algorithms L-shape, expanding square and level curves. By design, the sort-based algorithm never evaluates elements with cost higher than the optimal. Contour-line may evaluate elements higher than the optimal in rare occasions in the last iteration, but it did not happen in our experiments.

8. With matrix construction methods 1 and 2 sort-based algorithm and contour-line perform almost the exact same number of evaluations. The same is true for L-shape only for construction method 2.

Table 5 shows that when checking the oracle is computationally expensive, the sort-based algorithm and the contour-line algorithm are to be preferred. Sort-based algorithm guarantees the lowest number of oracle calls if there are no elements with the same cost as the optimal acceptable element. Contour-line, as shown in the previous tables, performs in practice around the same number of calls to the oracle. The algorithms are basically tied in all instances and are better than the rest.

For the large instances with a fast oracle the results are very different. In fact, the best algorithms are L-shape and expanding square while level curves also performs very well. Contour-line perform reasonably well when the optimal element is close to the main diagonal, but is always slower than expanding square. Observe that as noticed in the previous tables, L-shape is better when the optimal acceptable element is close to a border of the matrix while expanding square is better when the acceptable element is the main diagonal. Level curves performs reasonably well in most instances, outperforming L-shape when the optimal element is close to the main diagonal and expanding square when one of the coordinates is large and the other small. The best case for level curves is illustrated by instance i.20000.1000.20.0.0.0, when the algorithm finds the optimal element in the first iteration, while L-shape requires many, and the element is far from the diagonal, making expanding square waste time.
Table 5 Computational time for large instances.

<table>
<thead>
<tr>
<th>Instance</th>
<th>S-based</th>
<th>L-shape</th>
<th>C.-line</th>
<th>E. Square</th>
<th>Level curves</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.5000.5.5.0.0.1</td>
<td>0.002</td>
<td>1.546</td>
<td>0.002</td>
<td>0.002</td>
<td>0.414</td>
</tr>
<tr>
<td>I.5000.540.41.0.0.1</td>
<td>6.069</td>
<td>19.358</td>
<td>5.968</td>
<td>10.792</td>
<td>9.426</td>
</tr>
<tr>
<td>I.5000.541.541.0.0.1</td>
<td>22.220</td>
<td>189.546</td>
<td>22.627</td>
<td>22.456</td>
<td>60.554</td>
</tr>
<tr>
<td>I.5000.5.5.1.0.1</td>
<td>0.002</td>
<td>0.703</td>
<td>0.002</td>
<td>0.002</td>
<td>0.341</td>
</tr>
<tr>
<td>I.5000.41.500.1.0.1</td>
<td>6.141</td>
<td>7.129</td>
<td>6.109</td>
<td>9.600</td>
<td>7.090</td>
</tr>
<tr>
<td>I.5000.540.41.1.0.1</td>
<td>6.146</td>
<td>6.991</td>
<td>5.986</td>
<td>11.006</td>
<td>6.710</td>
</tr>
<tr>
<td>I.5000.541.541.1.0.1</td>
<td>22.698</td>
<td>34.331</td>
<td>21.984</td>
<td>22.033</td>
<td>30.392</td>
</tr>
<tr>
<td>I.20000.10000.10000.0.0.0</td>
<td>64.058</td>
<td>2.964</td>
<td>3.948</td>
<td>1.863</td>
<td>3.240</td>
</tr>
<tr>
<td>I.20000.10000.50.0.0.0</td>
<td>14.758</td>
<td>0.508</td>
<td>0.995</td>
<td>0.867</td>
<td>0.639</td>
</tr>
<tr>
<td>I.20000.5.10000.0.0.0</td>
<td>15.236</td>
<td>0.503</td>
<td>1.006</td>
<td>0.864</td>
<td>0.661</td>
</tr>
<tr>
<td>I.20000.250.250.0.0.0</td>
<td>0.026</td>
<td>0.097</td>
<td>0.003</td>
<td>0.001</td>
<td>0.008</td>
</tr>
<tr>
<td>I.20000.2400.10000.0.0.0</td>
<td>23.612</td>
<td>1.182</td>
<td>1.614</td>
<td>0.951</td>
<td>1.378</td>
</tr>
<tr>
<td>I.20000.2500.2500.0.0.0</td>
<td>3.411</td>
<td>1.009</td>
<td>0.329</td>
<td>0.136</td>
<td>0.482</td>
</tr>
<tr>
<td>I.20000.5000.5000.0.0.0</td>
<td>15.289</td>
<td>1.853</td>
<td>0.983</td>
<td>0.478</td>
<td>1.226</td>
</tr>
<tr>
<td>I.20000.1000.20.0.0.0</td>
<td>0.113</td>
<td>0.012</td>
<td>0.010</td>
<td>0.009</td>
<td>0.007</td>
</tr>
</tbody>
</table>

6 Conclusions

In this work we introduced a simple yet important problem in sorted matrices. The problem asks to find an optimal element of the matrix that satisfies a test performed by a black box oracle. The problem has applications in the assignment of resources to two tasks that must be performed simultaneously.

We showed that an algorithm using the sorted property of the matrix to call the oracle as few times as possible does not dominate in terms of computational complexity the naive algorithm consisting in calling the oracle for each element of the matrix. This observation motivated the development of other algorithms (L-shape, contour-line, expanding square and level curves) that do dominate the naive algorithm having the same complexity in the worst case but being strictly better in other cases.

Each algorithm was described and its complexity was analysed. Different properties including space requirements were discussed.

The algorithms were implemented and computational results were used to compare the number of evaluations of the oracle and the running time of each algorithm. By these experiments we were able to recommend different algorithms depending on the computational cost of the oracle, or on the position of the desired element, when prior knowledge is available.

An open question concerns the case of matrices $X + Y$ in which case it might be possible to use the special structure of those matrices to develop algorithms with lower computational complexity. Also, for some of the proposed algorithms it may be not obvious how to generalize them when the input matrix is not square. We plan to address this issue in a future work.

References

Finding the Minimum Cost Acceptable Element in a Sorted Matrix


