

1st Symposium on Algorithmic Foundations of Dynamic Networks

SAND 2022, March 28–30, 2022, Virtual Conference

Edited by

James Aspnes

Othon Michail



Editors

James Aspnes

Yale University, New Haven, Connecticut, USA
james.aspnes@gmail.com

Othon Michail 

University of Liverpool, UK
Othon.Michail@liverpool.ac.uk

ACM Classification 2012

Theory of computation; Mathematics of computing; Networks → Network algorithms

ISBN 978-3-95977-224-2

Published online and open access by

Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at <https://www.dagstuhl.de/dagpub/978-3-95977-224-2>.

Publication date

April, 2022

Bibliographic information published by the Deutsche Nationalbibliothek

The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at <https://portal.dnb.de>.

License

This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0):
<https://creativecommons.org/licenses/by/4.0/legalcode>.



In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:

- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.SAND.2022.0

LIPICs – Leibniz International Proceedings in Informatics

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

Editorial Board

- Luca Aceto (*Chair*, Reykjavik University, IS and Gran Sasso Science Institute, IT)
- Christel Baier (TU Dresden, DE)
- Mikolaj Bojanczyk (University of Warsaw, PL)
- Roberto Di Cosmo (Inria and Université de Paris, FR)
- Faith Ellen (University of Toronto, CA)
- Javier Esparza (TU München, DE)
- Daniel Král' (Masaryk University - Brno, CZ)
- Meena Mahajan (Institute of Mathematical Sciences, Chennai, IN)
- Anca Muscholl (University of Bordeaux, FR)
- Chih-Hao Luke Ong (University of Oxford, GB)
- 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)

ISSN 1868-8969

<https://www.dagstuhl.de/lipics>

■ Contents

Preface	
<i>James Aspnes and Othon Michail</i>	0:vii–0:viii
Organization	
.....	0:ix–0:x
Authors	
.....	0:xiii–0:xv

Invited Talks

Recent Advances in Fully Dynamic Graph Algorithms	
<i>Kathrin Hanauer, Monika Henzinger, and Christian Schulz</i>	1:1–1:47
Algorithmic Problems on Temporal Graphs	
<i>Paul G. Spirakis</i>	2:1–2:1
Networks, Dynamics, Algorithms, and Learning	
<i>Roger Wattenhofer</i>	3:1–3:1

Regular Papers

Atomic Splittable Flow Over Time Games	
<i>Antonia Adamik and Leon Sering</i>	4:1–4:16
Faster Exploration of Some Temporal Graphs	
<i>Duncan Adamson, Vladimir V. Gusev, Dmitriy Malyshev, and Viktor Zamaraev</i> .	5:1–5:10
Building Squares with Optimal State Complexity in Restricted Active Self-Assembly	
<i>Robert M. Alaniz, David Caballero, Sonya C. Cirlos, Timothy Gomez, Elise Grizzell, Andrew Rodriguez, Robert Schweller, Armando Tenorio, and Tim Wylie</i>	6:1–6:18
Loosely-Stabilizing Phase Clocks and The Adaptive Majority Problem	
<i>Petra Berenbrink, Felix Biermeier, Christopher Hahn, and Dominik Kaaser</i>	7:1–7:17
Complexity of Verification in Self-Assembly with Prebuilt Assemblies	
<i>David Caballero, Timothy Gomez, Robert Schweller, and Tim Wylie</i>	8:1–8:15
Robustness of Distances and Diameter in a Fragile Network	
<i>Arnaud Casteigts, Timothée Corsini, Hervé Hocquard, and Arnaud Labourel</i>	9:1–9:16
Computing Outside the Box: Average Consensus over Dynamic Networks	
<i>Bernadette Charron-Bost and Patrick Lambein-Monette</i>	10:1–10:16
Fast and Succinct Population Protocols for Presburger Arithmetic	
<i>Philipp Czerner, Roland Guttenberg, Martin Helfrich, and Javier Esparza</i>	11:1–11:17
Local Mutual Exclusion for Dynamic, Anonymous, Bounded Memory Message Passing Systems	
<i>Joshua J. Daymude, Andréa W. Richa, and Christian Scheideler</i>	12:1–12:19

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).

Editors: James Aspnes and Othon Michail



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Dynamic Size Counting in Population Protocols <i>David Doty and Mahsa Eftekhari</i>	13:1–13:18
Simulating 3-Symbol Turing Machines with SIMD DNA <i>David Doty and Aaron Ong</i>	14:1–14:15
Parameterized Temporal Exploration Problems <i>Thomas Erlebach and Jakob T. Spooner</i>	15:1–15:17
Bipartite Temporal Graphs and the Parameterized Complexity of Multistage 2-Coloring <i>Till Fluschnik and Pascal Kunz</i>	16:1–16:18
Temporal Connectivity: Coping with Foreseen and Unforeseen Delays <i>Eugen Fuchsle, Hendrik Molter, Rolf Niedermeier, and Malte Renken</i>	17:1–17:17
Fully Dynamic Four-Vertex Subgraph Counting <i>Kathrin Hanauer, Monika Henzinger, and Qi Cheng Hua</i>	18:1–18:17
Temporal Unit Interval Independent Sets <i>Danny Hermelin, Yuval Itzhaki, Hendrik Molter, and Rolf Niedermeier</i>	19:1–19:16
Search by a Metamorphic Robotic System in a Finite 3D Cubic Grid <i>Ryonosuke Yamada and Yukiko Yamauchi</i>	20:1–20:16

Brief Announcements

Brief Announcement: Cooperative Guarding in Polygons with Holes <i>John Augustine and Srikanth Ramachandran</i>	21:1–21:3
Brief Announcement: The Temporal Firefighter Problem <i>Samuel D. Hand, Jessica Enright, and Kitty Meeks</i>	22:1–22:3
Brief Announcement: Fault-Tolerant Shape Formation in the Amoebot Model <i>Irina Kostitsyna, Christian Scheideler, and Daniel Warner</i>	23:1–23:3
Brief Announcement: Barrier-1 Reachability for Thermodynamic Binding Networks Is PSPACE-Complete <i>Austin Luchsinger</i>	24:1–24:3

■ Preface

This volume contains the papers that were presented at the *1st Symposium on Algorithmic Foundations of Dynamic Networks*. Due to the COVID-19 pandemic, the conference was held online, March 28-30, 2022.

The Symposium on Algorithmic Foundations of Dynamic Networks (SAND) is a newly established conference. Its objective is to become the primary venue for original research on fundamental aspects of computing in dynamic networks and computational dynamics, bringing together researchers from computer science and related areas. SAND is seeking important contributions from all viewpoints, including theory and practice, characterized by a marked algorithmic aspect and addressing or being motivated by the role of dynamics in computing. It welcomes both conceptual and technical contributions, as well as novel ideas and new problems that will inspire the community and facilitate the further growth of the area.

The program committee of SAND 2022 consisted of James Aspnes (Co-Chair, Yale University), Luca Becchetti (University of Rome Sapienza), Petra Berenbrink (University of Hamburg), Janna Burman (Université Paris-Sud – LRI), Arnaud Casteigts (University of Bordeaux), Keren Censor-Hillel (Technion), Andrea Clementi (University of Rome Tor Vergata), Giuseppe Antonio Di Luna (University of Rome Sapienza), David Doty (University of California, Davis), Yuval Emek (Technion), Thomas Erlebach (Durham University), Sándor Fekete (TU Braunschweig), Paola Flocchini (University of Ottawa), David Ilcinkas (CNRS, Bordeaux), Zvi Lotker (Bar Ilan University), Toshimitsu Masuzawa (Osaka University), George Mertzios (Durham University), Othon Michail (Co-Chair, University of Liverpool), Rolf Niedermeier (TU Berlin), Rotem Oshman (Tel Aviv University), Andrea Richa (Arizona State University), Nicola Santoro (Carleton University), Christian Scheideler (University of Paderborn), David Soloveichik (University of Texas at Austin), Paul Spirakis (University of Liverpool and University of Patras), Damien Woods (Maynooth University), Viktor Zamaraev (University of Liverpool), and Christos Zaroliagis (University of Patras).

SAND 2022 received 30 submissions. The review process was double-blind and each paper was assigned to at least three members of the program committee with relevant expertise and eventually reviewed by them and/or by additional reviewers whenever needed. The program committee accepted 17 regular papers and 4 brief announcements that cover a wide range of topics in the broad area of algorithmic foundations of dynamic networks and computational dynamics, including DNA self-assembly, dynamic networks and distributed algorithms, mobile computing and robotics, population protocols, and temporal and dynamic graph algorithms. Keynote talks were given by distinguished researchers, to whom we are grateful: Monika Henzinger (University of Vienna), Paul Spirakis (University of Liverpool and University of Patras), and Roger Wattenhofer (ETH Zurich). We would also like to thank Kathrin Hanauer, Monika Henzinger, and Christian Schulz for contributing to the volume a survey on fully dynamic graph algorithms.

The program committee selected the paper “Fast and Succinct Population Protocols for Presburger Arithmetic” by Philipp Czerner, Roland Guttenberg, Martin Helfrich, and Javier Esparza for the Best Paper Award and the paper “Building Squares with Optimal State Complexity in Restricted Active Self-Assembly” by Robert M. Alaniz, David Caballero, Sonya C. Cirlos, Timothy Gomez, Elise Grizzell, Andrew Rodriguez, Robert Schweller, Armando Tenorio, and Tim Wylie for the Best Student Paper Award.

We wish to thank the members of the various committees of SAND as well as its advisory

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).

Editors: James Aspnes and Othon Michail

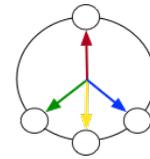


Leibniz International Proceedings in Informatics
LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

board, for all the hard work that they have put and which has made it possible to set up a new conference. All have been supportive throughout. We are grateful to the program committee members and to the additional reviewers for devoting time and effort in order to come up with a strong conference program. A special thanks goes to the general chairs of the organizing committee, Giuseppe Antonio Di Luna and Viktor Zamaraev. We are also indebted to the Chair of the SAND steering committee, Paola Flocchini, for all her support, to Giuseppe Prencipe for handling all the financial aspects, and to George Skretas for helping on publicity matters.

Above all, we thank the authors for submitting their work to SAND 2022. We can assure the reader that in this volume they will find well-presented ideas and results that make substantial contributions to our knowledge on the role of dynamics in computing. We do believe that this volume will inspire further work and will contribute to the further growth of this exciting research area.

Finally, we should point out that due to the outbreak of war in Ukraine, the SAND 2022 steering and organizing committees decided to replace the logo of the conference with the *peace dynamic graph*, until ceasefire and return to diplomacy and peace is achieved.



March, 2022

James Aspnes, *Yale University, USA*
Othon Michail, *University of Liverpool, UK*
SAND 2022 Program Chairs

■ Organization

Program Chairs

James Aspnes Yale University, USA
Othon Michail University of Liverpool, UK

Program Committee

James Aspnes (Co-Chair)	Yale University, USA
Luca Becchetti	University of Rome Sapienza, Italy
Petra Berenbrink	University of Hamburg, Germany
Janna Burman	Université Paris-Sud – LRI, France
Arnaud Casteigts	University of Bordeaux, France
Keren Censor-Hillel	Technion, Israel
Andrea Clementi	University of Rome Tor Vergata, Italy
Giuseppe Antonio Di Luna	University of Rome Sapienza, Italy
David Doty	University of California, Davis, USA
Yuval Emek	Technion, Israel
Thomas Erlebach	Durham University, UK
Sándor Fekete	TU Braunschweig, Germany
Paola Flocchini	University of Ottawa, Canada
David Ilcinkas	CNRS, Bordeaux, France
Zvi Lotker	Bar Ilan University, Israel
Toshimitsu Masuzawa	Osaka University, Japan
George Mertzios	Durham University, UK
Othon Michail (Co-Chair)	University of Liverpool, UK
Rolf Niedermeier	TU Berlin, Germany
Rotem Oshman	Tel Aviv University, Israel
Andrea Richa	Arizona State University, USA
Nicola Santoro	Carleton University, Canada
Christian Scheideler	University of Paderborn, Germany
David Soloveichik	University of Texas at Austin, USA
Paul Spirakis	University of Liverpool, UK and University of Patras, Greece
Damien Woods	Maynooth University, Ireland
Viktor Zamaraev	University of Liverpool, UK
Christos Zaroliagis	University of Patras, Greece

Organizing Committee

James Aspnes (Program Chair)	Yale University, USA
Giuseppe Antonio Di Luna (General Chair)	University of Rome Sapienza, Italy
Othon Michail (Program Chair)	University of Liverpool, UK
George Skretas (Publicity Chair)	University of Liverpool, UK
Viktor Zamaraev (General Chair)	University of Liverpool, UK

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).
Editors: James Aspnes and Othon Michail



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

0:x Organization

Steering Committee

James Aspnes (PC Chair 2022)	Yale University, USA
Giuseppe Antonio Di Luna (General Chair 2022)	University of Rome Sapienza, Italy
Paola Flocchini (Chair)	University of Ottawa, Canada
Othon Michail (PC Chair 2022)	University of Liverpool, UK
Giuseppe Prencipe (Treasurer)	Pisa University, Italy
Viktor Zamaraev (General Chair 2022)	University of Liverpool, UK

Advisory Board

James Aspnes	Yale University, USA
Luca Becchetti	University of Rome Sapienza, Italy
Arnaud Casteigts	University of Bordeaux, France
Giuseppe Antonio Di Luna	University of Rome Sapienza, Italy
Paola Flocchini	University of Ottawa, Canada
George Mertzios	Durham University, UK
Othon Michail	University of Liverpool, UK
Rolf Niedermeier	TU Berlin, Germany
Rotem Oshman	Tel Aviv University, Israel
Nicola Santoro	Carleton University, Canada
Paul Spirakis	University of Liverpool, UK and University of Patras, Greece
Viktor Zamaraev	University of Liverpool, UK

Additional Reviewers

Duncan Adamson	Abdul Ghani	Mina Latifi
Matthias Bentert	Thorsten Götte	Andreas Padalkin
Joseph Briones	Luciano Gualà	Francesco Pasquale
Timothée Corsini	Klaus Heeger	Josh Petrack
Francesco d'Amore	Kristian Hinnenthal	Christoforos Raptopoulos
Joshua Daymude	Nina Klobas	Arne Schmidt
Fabien Dufoulon	Irina Kostitsyna	George Skretas
Mahsa Eftekhari Hesari	Pascal Kunz	Michail Theofilatos

Supporters

SAND 2022 would like to thank the School of EEE/CS and the Department of Computer Science of the University of Liverpool, the Department of Computer Science of the University of Pisa, and the Sapienza University of Rome for their support. SAND 2022 was also made possible by the use of EasyChair as the submission server and review process management system, due to LIPIcs producing and publishing the proceedings, Zoom which was used as the video conferencing system, and Gather used for breaks and socializing.



SAPIENZA
UNIVERSITÀ DI ROMA

■ List of Authors

Antonia Adamik (4)
Technische Universität Berlin, Germany

Duncan Adamson (5)
Department of Computer Science, Reykjavik
University, Iceland

Robert M. Alaniz (6)
Department of Computer Science, University of
Texas Rio Grande Valley, TX, USA

John Augustine  (21)
Department of Computer Science & Engineering,
Indian Institute of Technology Madras, India

Petra Berenbrink (7)
Universität Hamburg, Germany

Felix Biermeier (7)
Universität Hamburg, Germany

David Caballero (6, 8)
Department of Computer Science, University of
Texas Rio Grande Valley, TX, USA

Arnaud Casteigts  (9)
LaBRI, CNRS, Université de Bordeaux,
Bordeaux INP, France

Bernadette Charron-Bost (10)
Département d'informatique de l'ENS, ENS,
CNRS, PSL University, Paris, France

Sonya C. Cirlos (6)
Department of Computer Science, University of
Texas Rio Grande Valley, TX, USA

Timotheé Corsini  (9)
LaBRI, CNRS, Université de Bordeaux,
Bordeaux INP, France

Philipp Czerner  (11)
Department of Informatics, Technische
Universität München, Germany

Joshua J. Daymude  (12)
Biodesign Center for Biocomputing, Security
and Society, Arizona State University, Tempe,
AZ, USA

David Doty  (13, 14)
University of California, Davis, CA, USA

Mahsa Eftekhari  (13)
University of California, Davis, CA, USA

Jessica Enright  (22)
School of Computing Science, University of
Glasgow, UK

Thomas Erlebach  (15)
Department of Computer Science, Durham
University, UK

Javier Esparza  (11)
Department of Informatics, Technische
Universität München, Germany

Till Fluschnik  (16)
Algorithmics and Computational Complexity,
Technische Universität Berlin, Germany

Eugen Füchsle (17)
Faculty IV, Algorithmics and Computational
Complexity, TU Berlin, Germany

Timothy Gomez (6, 8)
Department of Computer Science, University of
Texas Rio Grande Valley, TX, USA

Elise Grizzell (6)
Department of Computer Science, University of
Texas Rio Grande Valley, TX, USA

Vladimir V. Gusev (5)
Materials Innovation Factory, University of
Liverpool, UK; Department of Computer
Science, University of Liverpool

Roland Guttenberg  (11)
Department of Informatics, Technische
Universität München, Germany

Christopher Hahn (7)
Universität Hamburg, Germany

Kathrin Hanauer  (1, 18)
Faculty of Computer Science, Universität Wien,
Austria

Samuel D. Hand  (22)
School of Computing Science, University of
Glasgow, UK

Martin Helfrich  (11)
Department of Informatics, Technische
Universität München, Germany

Monika Henzinger  (1, 18)
Faculty of Computer Science, Universität Wien,
Austria

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).
Editors: James Aspnes and Othon Michail



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

- Danny Hermelin (19)
Department of Industrial Engineering and Management, Ben-Gurion University of the Negev, Beer-Sheva, Israel
- Hervé Hocquard  (9)
LaBRI, CNRS, Université de Bordeaux, Bordeaux INP, France
- Qi Cheng Hua (18)
Faculty of Computer Science, University of Vienna, Austria
- Yuval Itzhaki (19)
Faculty IV, Algorithmics and Computational Complexity, TU Berlin, Germany
- Dominik Kaaser  (7)
Universität Hamburg, Germany
- Irina Kostitsyna  (23)
Department of Mathematics and Computer Science, Eindhoven University of Technology, The Netherlands
- Pascal Kunz  (16)
Algorithmics and Computational Complexity, Technische Universität Berlin, Germany
- Arnaud Labourel  (9)
Aix Marseille Univ, CNRS, LIS, Marseille, France
- Patrick Lambein-Monette  (10)
Université Paris Cité, CNRS, IRIF, F-75013, Paris, France
- Austin Luchsinger (24)
The University of Texas at Austin, TX, USA
- Dmitriy Malyshev (5)
Laboratory of Algorithms and Technologies for Network Analysis, HSE University, Nizhny Novgorod, Russian Federation
- Kitty Meeks  (22)
School of Computing Science, University of Glasgow, UK
- Hendrik Molter  (17, 19)
Department of Industrial Engineering and Management, Ben-Gurion University of the Negev, Beer-Sheva, Israel
- Rolf Niedermeier  (17, 19)
Faculty IV, Algorithmics and Computational Complexity, TU Berlin, Germany
- Aaron Ong (14)
University of California, Davis, CA, USA
- Srikanth Ramachandran  (21)
Department of Computer Science & Engineering, Indian Institute of Technology Madras, India
- Malte Renken  (17)
Faculty IV, Algorithmics and Computational Complexity, TU Berlin, Germany
- Andréa W. Richa  (12)
School of Computing and Augmented Intelligence, Arizona State University, Tempe, AZ, USA
- Andrew Rodriguez (6)
Department of Computer Science, University of Texas Rio Grande Valley, TX, USA
- Christian Scheideler  (12, 23)
Department of Computer Science, Universität Paderborn, Germany
- Christian Schulz  (1)
Faculty of Mathematics and Computer Science, Universität Heidelberg, Germany
- Robert Schweller (6, 8)
Department of Computer Science, University of Texas Rio Grande Valley, TX, USA
- Leon Sering  (4)
ETH Zürich, Switzerland
- Paul G. Spirakis  (2)
Department of Computer Science, University of Liverpool, UK; Computer Engineering & Informatics Department, University of Patras, Greece
- Jakob T. Spooner  (15)
School of Computing and Mathematical Sciences, University of Leicester, UK
- Armando Tenorio (6)
Department of Computer Science, University of Texas Rio Grande Valley, TX, USA
- Daniel Warner  (23)
Department of Computer Science, Paderborn University, Germany
- Roger Wattenhofer (3)
ETH Zürich, Switzerland
- Tim Wylie (6, 8)
Department of Computer Science, University of Texas Rio Grande Valley, TX, USA
- Ryonosuke Yamada (20)
Graduate School of Information Science and Electrical Engineering, Kyushu University, Fukuoka, Japan

Yukiko Yamauchi (20)
Faculty of Information Science and Electrical
Engineering, Kyushu University, Fukuoka, Japan

Viktor Zamaraev (5)
Department of Computer Science, University of
Liverpool, UK

Recent Advances in Fully Dynamic Graph Algorithms

Kathrin Hanauer ✉ 

Faculty of Computer Science, Universität Wien, Austria

Monika Henzinger ✉ 

Faculty of Computer Science, Universität Wien, Austria

Christian Schulz ✉ 

Faculty of Mathematics and Computer Science, Universität Heidelberg, Germany

Abstract

In recent years, significant advances have been made in the design and analysis of fully dynamic algorithms. However, these theoretical results have received very little attention from the practical perspective. Few of the algorithms are implemented and tested on real datasets, and their practical potential is far from understood. Here, we present a quick reference guide to recent engineering and theory results in the area of fully dynamic graph algorithms.

2012 ACM Subject Classification General and reference → Surveys and overviews; Networks → Network dynamics; Mathematics of computing → Graph algorithms

Keywords and phrases fully dynamic graph algorithms, survey

Digital Object Identifier 10.4230/LIPIcs.SAND.2022.1

Category Invited Talk

Funding This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (Grant agreement No. 101019564, “The Design of Modern Fully Dynamic Data Structures (MoDynStruct)”), as well as from the Austrian Science Fund (FWF) and netIDEE SCIENCE project P 33775-N. Moreover, we have been partially supported by DFG grant SCHU 2567/1-2.



1 Introduction

A (fully) dynamic graph algorithm is a data structure that supports edge insertions, edge deletions, and answers certain queries that are specific to the problem under consideration. There has been a lot of research on dynamic algorithms for graph problems that are solvable in polynomial time by a static algorithm. The most studied dynamic problems are graph problems such as connectivity, reachability, shortest paths, or matching (see [115]). Typically, any dynamic algorithm that can handle edge insertions can be used as a static algorithm by starting with an empty graph and inserting all m edges of the static input graph step-by-step. A fundamental question that arises is which problems can be *fully dynamized*, which boils down to the question whether they admit a dynamic algorithm that supports updates in $\mathcal{O}(T(m)/m)$ time, where $T(m)$ is the static running time. Thus, for static problems that can be solved in near-linear time, the research community is interested in near-constant time updates. By now, such results have been achieved for a wide range of problems [115], which resulted in a rich algorithmic toolbox spanning a wide range of techniques. However, while there is a large body of theoretical work on efficient dynamic graph algorithms, until recently there has been very little on their empirical evaluation. For some classical dynamic algorithms, experimental studies have been performed, such as for fully dynamic graph clustering [76] and fully dynamic approximation of betweenness centrality [33]. However, for



© Kathrin Hanauer, Monika Henzinger, and Christian Schulz;
licensed under Creative Commons License CC-BY 4.0

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).

Editors: James Aspnes and Othon Michail; Article No. 1; pp. 1:1–1:47

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

other fundamental dynamic graph problems, the theoretical algorithmic ideas have received very little attention from the practical perspective. In particular, very little work has been devoted to engineering such algorithms and providing efficient implementations in practice. Previous surveys on the topic [249, 10] are more than twenty years old and do not capture the state-of-the-field anymore. In this work, we aim to survey recent progress in theory as well as in the empirical evaluation of fully dynamic graph algorithms and summarize methodologies used to evaluate such algorithms. Moreover, we point to theoretical results that we think have a good potential for practical implementations. Hence, this paper should help an unfamiliar reader by providing most recent references for various problems in fully dynamic graph algorithms. Lastly, there currently is a lack of fully dynamic real-world graphs available online – most of the instances that can be found to date are insertions-only. Hence, together with this survey we will also start a new open-access graph repository that provides fully dynamic graph instances^{1,2}.

We want to point out that there are also various dynamic graph models which we cannot discuss in any depth for space limitations. These are insertions-only algorithms, deletions-only algorithms, offline dynamic algorithms, algorithms with vertex insertions and deletions, kinetic algorithms, temporal algorithms, algorithms with a limit on the number of allowed queries, algorithms for the sliding-windows model, and algorithms for sensitivity problems (also called emergency planning or fault-tolerant algorithms). We also exclude dynamic algorithms in other models of computation such as distributed algorithms and algorithms in the massively parallel computation (MPC) model. If the full graph is known at preprocessing time and vertices are “switched on and off”, this is called the *subgraph model*, whereas *algorithms under failures* deal with the case that vertices or edges are only “switched off”. We do not discuss these algorithms either.

Note that fully dynamic graph algorithms (according to our definition) are also sometimes called *algorithms for evolving graphs* or *for incremental graphs* or sometimes even *maintaining a graph online*.

2 Preliminaries

Let $G = (V, E)$ be a (un)directed graph with vertex set V and edge set E . Throughout this paper, let $n = |V|$ and $m = |E|$. The *density* of G is $d = \frac{m}{n}$. In the directed case, an edge $(u, v) \in E$ has *tail* u and *head* v and u and v are said to be *adjacent*. (u, v) is said to be an *outgoing* edge or *out-edge* of u and an *incoming* edge or *in-edge* of v . The *outdegree* $\deg^+(v)$ /*indegree* $\deg^-(v)$ /*degree* $\deg(v)$ of a vertex v is its number of (out-/in-) edges. The *out-neighborhood* (*in-neighborhood*) of a vertex u is the set of all vertices v such that $(u, v) \in E$ ($(v, u) \in E$). In the undirected case, $N(v) := \{u : \{v, u\} \in E\}$ denotes the *neighbors* of v . The degree of a vertex v is $\deg(v) := |N(v)|$ here. In the following, Δ denotes the maximum degree that can be found in any state of the dynamic graph. Our focus in this paper are *fully dynamic graphs*, where the number of vertices is fixed, but edges can be added and removed. We use $\tilde{O}(\cdot)$ to hide polylogarithmic factors.

¹ If you have access to fully dynamic instances, we are happy to provide them in our repository.

² <https://DynGraphLab.github.io>

2.1 Conditional Lower Bounds

There are lower bounds for fully dynamic graph algorithms based on various popular conjectures initiated by [183, 3, 117]. These lower bounds usually involve three parameters: the preprocessing time $p(m, n)$, the update time $u(m, n)$, and the query time $q(m, n)$. We will use the notation $(p(m, n), u(m, n), q(m, n))$ below to indicate that no algorithm with preprocessing time at most $p(m, n)$ exists that requires at most update time $u(m, n)$ and query time $q(m, n)$. Note that if the preprocessing time is larger than $p(m, n)$ or if the query time is larger than $q(m, n)$, then it might be possible to achieve an update time better than $u(m, n)$. In the same vein, if the preprocessing time is larger than $p(m, n)$ or if the update time is larger than $u(m, n)$, then it might be possible to achieve a query time better than $q(m, n)$. We will write *poly* to denote any running time that is *polynomial* in the size of the input.

Any conditional lower bound that is based on the OMv (Online Boolean Matrix-Vector Multiplication) conjecture [117] applies to both the (amortized or worst-case) running time of any fully dynamic algorithm *and also* to the worst-case running time of insertions-only and deletions-only algorithms. We will not mention this for each problem below and only state the lower bound, except in cases where as a result of the lower bound only algorithms for the insertions-only or deletions-only setting have been studied.

3 Fully Dynamic Graph Algorithms

In this section, we describe recent efforts in fully dynamic graph algorithms. We start by describing fundamental problems that we think belong to a basic toolbox of fully dynamic graph algorithms: strongly connected components, minimum spanning trees, cycle detection/topological ordering, matching, core decomposition, subgraph detection, diameter, as well as independent sets. Later on, we discuss problems that are closer to the application side. To this end we include fully dynamic algorithms for shortest paths, maximum flows, graph clustering, centrality measures, and graph partitioning.

3.1 (Strongly) Connected Components and BFS/DFS Trees

One of the most fundamental questions on graphs is whether two given vertices are connected by a path. In the undirected case, a path connecting two vertices u and w is a sequence of edges $\mathcal{P} = (\{u, v_0\}, \{v_0, v_1\}, \dots, \{v_k, w\})$. A *connected component* is a maximal set of vertices that are pairwise connected by a path. A graph is *connected* if there is exactly one connected component, which is V . In a directed graph, we say that a vertex u can *reach* a vertex w if there is a directed path from u to w , i.e., a sequence of directed edges $\mathcal{P} = ((u, v_0), (v_0, v_1), \dots, (v_k, w))$. A *strongly connected component* (SCC) is a maximal set of vertices that can reach each other pairwise. A directed graph is *strongly connected* if there is just one strongly connected component, which is V . The *transitive closure* of a graph G is a graph on the same vertex set with an edge $(u, w) \in V \times V$ if and only if u can reach w in G . Given an undirected graph, we can construct a directed graph from it by replacing each undirected edge $\{u, w\}$ by a pair of directed edges (u, w) and (w, u) and translate queries of connectedness into reachability queries on the directed graph. A breadth-first search (BFS) or depth-first search (DFS) traversal of a directed or undirected graph defines a rooted, spanning subtree that consists of the edges via which a new vertex was discovered. Apart from connectivity or reachability, BFS and DFS trees can be used to answer a variety of problems on graphs, such as testing bipartiteness, shortest paths in the unweighted setting, 2-edge connectivity, or biconnectivity.

Undirected Graphs (Connectivity)

Patrascu and Demaine [184] gave an (unconditional) lower bound of $\Omega(\log n)$ per operation for this problem, improving a bound of $\Omega(\log n / \log \log n)$ [123]. The first non-trivial dynamic algorithms for connectivity, and also for 2-edge connectivity, and 2-vertex connectivity [86, 121, 79, 80, 122] took time $\tilde{O}(\sqrt{n})$ per operation. Henzinger and King [125] were the first to give a fully dynamic algorithm with polylogarithmic time per operation for this problem. Their algorithm is, however, randomized. Holm et al. [127] gave the first deterministic fully dynamic algorithm with polylogarithmic time per operation. The currently fastest fully dynamic connectivity algorithm takes $\mathcal{O}(\log n (\log \log n)^2)$ amortized expected time per operation [132]. There also is a batch-dynamic parallel algorithm that answers k queries in $\mathcal{O}(k \log(1 + n/k))$ expected work and $\mathcal{O}(\log n)$ depth with $\mathcal{O}(\log n \log(1 + n/B))$ expected amortized work per update and $\mathcal{O}(\log^3 n)$ depth for an average batch size of B [6].

The fully dynamic connectivity problem can be reduced to the maintenance of a spanning forest, using, e.g., dynamic trees [222, 7] or Euler tour trees [124, 232] (see also Section 3.2), for the components. If the graph is a forest, updates and queries can be processed in amortized $\mathcal{O}(\log n)$ time, whereas the theoretically fastest algorithms [141] to date for general graphs have polylogarithmic worst-case update time and $\mathcal{O}(\log n / \log \log n)$ worst-case query time, the latter matching the lower bound [123, 168]. The key challenge on general graphs is to determine whether the deletion of an edge of the spanning forest disconnects the component or whether a replacement edge can be found. There are also fully dynamic algorithms for more refined notions of connectivity: Two-edge connectivity [125, 126] and two-vertex connectivity [126] can also be maintained in polylogarithmic time per operation. See [134] for a survey on that topic.

Building on an earlier study by Alberts et al. [10], Iyer et al. [137] experimentally compared the Euler tour tree-based algorithms by Henzinger and King [124] and Holm et al. [126] to each other as well as several heuristics to achieve speedups in both candidates. The instances used in the evaluation were random graphs with random edge insertions and deletions, random graphs where a fixed set of edges appear and disappear dynamically, graphs consisting of cliques of equal size plus a set of inter-clique edges, where only the latter are inserted and deleted, as well as specially crafted worst-case instances for the algorithms. The authors showed that the running time of both algorithms can be improved distinctly via heuristics; in particular a sampling approach to replace deleted tree edges has proven to be successful. The experimental running time of both algorithms was comparable, but with the heuristics, the algorithm by Holm et al. [126] performed better.

Baswana et al. [25] gave the first algorithm for maintaining an undirected DFS tree with $o(m)$ update time and showed a conditional lower bound of $\Omega(n)$ on the update time in case of vertex updates and, if the tree is maintained explicitly, an unconditional lower bound of $\Omega(n)$ under edge updates. Their algorithm has a preprocessing time of $\mathcal{O}(m \log n)$, a worst-case update time of $\mathcal{O}(\sqrt{mn} \log^{2.5} n)$, and uses $\mathcal{O}(m \log^2 n)$ bits. Nakamura and Sadakane [172] improved the update time by polylog n factors and the space required to $\mathcal{O}(m \log n)$. Recently, Baswana et al. [27] further reduced the update time down to $\mathcal{O}(\sqrt{mn} \log n)$. A parallel algorithm that uses m processors and $\mathcal{O}(\text{polylog } n)$ update time was given by Khan [145]. To the best of our knowledge, experimental evaluations have only been conducted to date with algorithms designed for the *incremental* setting, but not for fully-dynamic algorithms. No experimental studies on dynamically maintaining BFS trees are known to us.

Directed Graphs (Reachability, Strong Connectivity, Transitive Closure)

For directed graphs that are and remain acyclic, the same algorithms can be employed for reachability as for (undirected) connectivity in forests (see above). On general graphs, there is a conditional lower bound of $(\text{poly}, m^{1/2-\delta}, m^{1-\delta})$ for any small constant $\delta > 0$ based on the OMv conjecture. This bound even holds for the s - t reachability problem, where both s and t are fixed for all queries. The currently fastest algorithms for transitive closure are three Monte Carlo algorithms with one-sided error: Two by Sankowski [205] with $\mathcal{O}(1)$ or $\mathcal{O}(n^{0.58})$ worst-case query time and $\mathcal{O}(n^2)$ or $\mathcal{O}(n^{1.58})$ worst-case update time, respectively, and one by van den Brand, Nanongkai, and Saranurak [234] with $\mathcal{O}(n^{1.407})$ worst-case update and worst-case query time. There exists a conditional lower bound based on a variant of the OMv conjecture that shows that these running times are optimal [234]. Moreover, there are two deterministic, combinatorial algorithms: Roditty’s algorithm with constant query time and $\mathcal{O}(n^2)$ amortized update time [198], as well as one by Roditty and Zwick [201] with an improved $\mathcal{O}(m + n \log n)$ amortized update time at the expense of $\mathcal{O}(n)$ worst-case query time.

Frigioni et al. [89] and later Krommidas and Zaroliagis [153] empirically studied the performance of an extensive number of algorithms for transitive closure, including those mentioned above. They also developed various extensions and variations and compared them not only to each other, but also to static, so-called “simple-minded” algorithms such as breadth-first and depth-first search. Their evaluation included random Erdős-Renyí graphs, specially constructed hard instances, as well as two instances based on real-world graphs. It showed that the “simple-minded” algorithms could outperform the dynamic ones distinctly and up to several factors, unless the query ratio was more than 65 % or the instances were dense random graphs.

In recent experimental studies by Hanauer et al. [110, 109], two relatively straightforward algorithms for single-source reachability could outperform the “simple-minded” algorithms of the earlier studies in a single-source setting by several orders of magnitude in practice both on random graphs as well as on real-world instances: SI maintains an arbitrary reachability tree which is re-constructed via a combined forward and backward breadth-first search traversal on edge deletions if necessary and is especially fast if insertions predominate, which can be handled in $\mathcal{O}(n + m)$ time. By contrast, it may take up to $\mathcal{O}(nm)$ time for a single edge removal. SES is an extension and simplification of Even-Shiloach trees [220], which originally only handle edge deletions. Its strength are hence instances with many deletions. As a plus, it is able to deliver not just any path as a witness for reachability, but even the shortest path (with respect to the number of edges). Furthermore, it internally maintains a BFS tree, which makes it viable also for numerous other applications, see above. Its worst-case update time is $\mathcal{O}(n + m)$, and, like SI, it answers queries in constant time. One key ingredient for the superior performance of both algorithms in practice are carefully chosen criteria for an abortion of the re-construction of their data structures and their re-building from scratch [110]. To query the transitive closure of a graph, a number of so-called “supportive vertices”, for which both in- and out-reachability trees are maintained explicitly, can be picked either once or periodically anew and then be used to answer both positive and negative reachability queries between a number of pairs of vertices decisively in constant time [109]. The fallback routine can be a simple static graph traversal and therefore be relatively expensive: With a random initial choice of supportive vertices and no periodic renewals, this approach has been shown to answer a great majority of reachability queries on both random and real-world instances in constant time already if the number of supportive vertices is very small, i.e., two or three.

These experimental studies clearly show the limitations of worst-case analysis: All implemented algorithms are fully dynamic with at least linear worst-case running time per operation and, thus, all perform “(very) poor” in the worst case. Still on all graphs used in the study the relatively simple new algorithms clearly outperformed the algorithms used in previous studies.

Yang et al. [246] were the first to give a fully dynamic algorithm for maintaining a DFS tree in a directed graph along with several optimizations to achieve speedups in practice. In an experimental evaluation on twelve real-world instances, they showed that the optimized version of their algorithm can handle edge insertions and deletions within few seconds on average for instances with millions of vertices. With regard to BFS trees, the already mentioned SES algorithm [110] is the only fully dynamic algorithm we are aware of that maintains a BFS tree on a directed graph.

3.2 Minimum Weight Spanning Trees

A minimum weight spanning tree (MST) of a connected graph is a subset of the edges such that all nodes are connected via the edges in the subset, the induced subgraph has no cycles and, lastly, has the minimum total weight among all possible subsets fulfilling the first two properties.

The lower bound of $\Omega(\log n)$ [184] on the time per operation for connectivity trivially extends to maintaining the weight of a minimum spanning tree. Holm et al. [127] gave the first fully dynamic algorithm with polylogarithmic time per operation for this problem. It was later slightly improved to $\mathcal{O}(\log^4 n) / \log \log n$ time per operation [128].

Amato et al. [133] presented the first experimental study of dynamic minimum spanning tree algorithms. In particular, the authors implemented different versions of Frederickson’s algorithm [85] which uses partitions and topology trees. The algorithms have been adapted with sparsification techniques to improve their performance. The update running times of these algorithms range from $\mathcal{O}(m^{2/3})$ to $\mathcal{O}(m^{1/2})$. The authors further presented a variant of Frederickson’s algorithm that is significantly faster than all other implementations of this algorithm. However, the authors also proposed a simple adaption of a partially dynamic data structure of Kruskal’s algorithm that was the fastest implementation on random inputs. Later, Cattaneo et al. [56, 57] presented an experimental study on several algorithms for the problem. The authors presented an efficient implementation of the algorithm of Holm et al. [127], proposed new simple algorithms for dynamic MST that are not as asymptotically efficient as the algorithm by Holm et al. but seem to be fast in practice, and lastly compared their algorithms with the results of Amato et al. [133]. The algorithm by Holm et al. uses a clever refinement of a technique by Henzinger and King [119] for developing fully dynamic algorithms starting from the deletions-only case. One outcome of their experiments is that simple algorithms outperform the theoretically more heavy algorithms on random and worst-case networks. On the other hand, on k -clique inputs, i.e. graphs that contain k cliques of size c plus $2k$ randomly chosen inter-clique edges, the implementation of the algorithm by Holm et al. outperformed the simpler algorithms.

Tarjan and Werneck [227] performed experiments for several variants of dynamic trees data structure. The evaluated data structures have been used by Ribero and Toso [196], who focused on the case of changing weights, i.e. the edges of the graph are constant, but the edge weights can change dynamically. The authors also proposed and used a new data structure for dynamic tree representation called DRD-trees. In their algorithm the dynamic tree data structure is used to speed up connectivity queries that check whether two vertices belong to different subtrees. More generally, the authors compared different types of data structures

to do this task. In particular, the authors used the dynamic tree data structures that have been evaluated by Tarjan and Werneck [227]. The experimental evaluation demonstrated that the new structure reduces the computation time observed for the algorithm of Cattaneo et al. [56], and at the same time yielded the fastest algorithms in the experiments.

3.3 Cycle Detection and Topological Ordering

A cycle in a (directed) graph $G = (V, E)$ is a non-empty path $\mathcal{P} = (v_1, \dots, v_k = v_1)$ such that $(v_i, v_{i+1}) \in E$. A topological ordering of a directed graph is a linear ordering of its vertices from 1 to n such that for every directed edge (u, v) from vertex u to vertex v , u is ordered before v . In the static case, one can use a depth-first search (DFS) to compute a topological ordering of a directed acyclic graph or to check if a (un)directed graph contains a cycle.

Let $\delta > 0$ be any small constant. Based on the OMv conjecture [117] it is straightforward to construct a lower bound of $(\text{poly}, m^{1/2-\delta}, m^{1-\delta})$ for the (amortized or worst-case) running time of any fully dynamic algorithm that detects whether the graph contains any cycle. As any algorithm for topological ordering can be used to decide whether a graph contains a cycle, this lower bound also applies to any fully dynamic topological ordering algorithm. Via dynamic matrix inverse one can maintain fully dynamic directed cycle detection in $\mathcal{O}(n^{1.407})$ [234], which is conditionally optimal based on a variant of the OMv conjecture.

Pearce and Kelly [187, 188] were the first to evaluate algorithms for topological ordering in the presence of edge insertions and deletions. In their work, the authors compared three algorithms that can deal with the online topological ordering problem. More precisely, the authors implemented the algorithms by Marchetti-Spaccamela et al. [164] and Alpern et al. [12] as well as a newly developed algorithm. Their new algorithm is the one that performed best in their experiments. The algorithm maintains a node-to-index map, called $n2i$, that maps each vertex to a unique integer in $\{1 \dots n\}$ and ensures that for any edge (u, v) in G , it holds $n2i[u] < n2i[v]$. When an insertion (u, v) invalidates the topological ordering, affected nodes are updated. The set of affected nodes are identified using a forward DFS from v and backward DFS from u . The two sets are then separately sorted into increasing topological order and afterwards a remapping to the available indices is performed. The algorithm by Marchetti-Spaccamela et al. [164] is quite similar to the algorithm by Pearce and Kelly. However, it only maintains the forward set of affected nodes and obtains a correct solution by shifting the affected nodes up in the ordering (putting them after u). Alpern et al. [12] used a data structure to create new priorities between existing ones in constant worst-case time. The result by Pearce and Kelly has later been applied to online cycle detection and difference propagation in pointer analysis by Pearce et al. [189]. Furthermore, Pearce and Kelly [186] later extended their algorithm to be able to provide more efficient batch updates.

3.4 (Weighted) Matching

The matching problem is one of the most prominently studied combinatorial graph problems having a variety of practical applications. A matching \mathcal{M} of a graph $G = (V, E)$ is a subset of edges such that no two elements of \mathcal{M} have a common end point. Many applications require matchings with certain properties, like being maximal (no edge can be added to \mathcal{M} without violating the matching property) or having maximum cardinality.

In the dynamic setting, there is a conditional lower bound of $(\text{poly}, m^{1/2-\delta}, m^{1-\delta})$ (for any small constant $\delta > 0$) for the size of the maximum cardinality matching based on the OMv conjecture [117]. Of course, maintaining an actual maximum matching is only harder than

maintaining the size of a maximum matching. Thus upper bounds have mostly focused on approximately maximum matching. However, also here we have to distinguish (a) algorithms that maintain the *size* of an approximately maximum matching and (b) algorithms that maintain an *approximately maximum matching*.

- (a) Improving Sankowski's $O(n^{1.495})$ update time bound [207], van den Brand et al. [234] maintain the exact size of a maximum matching in $O(n^{1.407})$ update time. To maintain the approximate size of the maximum matching, dynamic algorithms use the duality of maximum matching and vertex cover and maintain instead a $(2 + \epsilon)$ -approximate vertex cover. This line of work led to a sequence of papers [135, 40, 42, 39], resulting finally in a deterministic $(2 + \epsilon)$ -approximation algorithm that maintains a hierarchical graph decomposition with $O(1/\epsilon^2)$ amortized update time [47]. The algorithm can be turned into an algorithm with worst-case $O(\log^3 n)$ time per update [43].
- (b) One can trivially maintain a maximal matching in $O(n)$ update time by resolving all trivial augmenting paths, i.e. cycle-free paths that start and end on a unmatched vertex and where edges from \mathcal{M} alternate with edges from $E \setminus \mathcal{M}$, of length one. As any maximal matching is a 2-approximation of a maximum matching, this leads to a 2-approximation algorithm. Onak and Rubinfeld [181] presented a randomized algorithm for maintaining an $O(1)$ -approximate matching with $O(\log^2 n)$ expected amortized time per edge update. Baswana, Gupta, and Sen [26] gave an elegant algorithm that maintains a *maximal* matching with amortized update time $O(\log n)$. It is based on a hierarchical graph decomposition and was subsequently improved by Solomon to amortized constant expected update time [223]. For worst-case bounds, the best results are a $(1 + \epsilon)$ -approximation in $O(\sqrt{m}/\epsilon)$ update time by Gupta and Peng [104] (see [178] for a 3/2-approximation in the same time), a $(3/2 + \epsilon)$ -approximation in $O(m^{1/4}/\epsilon^{2.5})$ time by Bernstein and Stein [37], and a $(2 + \epsilon)$ -approximation in $O(\text{polylog } n)$ time by Charikar and Solomon [59] and Arar et al. [17]. Recently, Grandoni et al. [100] gave an incremental matching algorithm that achieves a $(1 + \epsilon)$ -approximate matching in constant deterministic amortized time. Finally, Bernstein et al. [36] improved the maximal matching algorithm of Baswana et al. [26] to $O(\log^5 n)$ worst-case time with high probability.

Despite this variety of different algorithms, to the best of our knowledge, there have been only limited efforts so far to engineer and evaluate these algorithms on real-world instances. Henzinger et al. [116] initiated the empirical evaluation of algorithms for this problem in practice. To this end, the authors evaluated several dynamic maximal matching algorithms as well as an algorithm that is able to maintain the maximum matching. They implemented the algorithm by Baswana, Gupta and Sen [26], which performs edge updates in $O(\sqrt{n})$ time and maintains a 2-approximate maximum matching, the algorithm of Neiman and Solomon [178], which takes $O(\sqrt{m})$ time to maintain a 3/2-approximate maximum matching, as well as two novel dynamic algorithms, namely a random walk-based algorithm as well as a dynamic algorithm that searches for augmenting paths using a (depth-bounded) blossom algorithm. Their experiments indicate that an optimum matching can be maintained dynamically more than an order of magnitude faster than the naive algorithm that recomputes maximum matchings from scratch. Second, all non-optimum dynamic algorithms that have been considered in this work were able to maintain near-optimum matchings in practice while being multiple orders of magnitudes faster than the naive exact dynamic algorithm. The study concludes that in practice an extended random walk-based algorithm is the method of choice.

For the *weighted* dynamic matching problem, Anand et al. [14] proposed an algorithm that can maintain an 4.911 -approximate dynamic maximum weight matching that runs in amortized $\mathcal{O}(\log n \log C)$ time where C is the ratio of the weight of the highest weight edge to the weight of the smallest weight edge. Furthermore, a sequence [41, 1, 39, 46, 44] of work on fully dynamic set cover resulted in $(1 + \epsilon)$ -approximate weighted dynamic matching algorithms, with $\mathcal{O}(1/\epsilon^3 + (1/\epsilon^2) \log C)$ amortized and $\mathcal{O}((1/\epsilon^3) \log^2(Cn))$ worst-case time per operation based on various hierarchical hypergraph decompositions. Gupta and Peng [105] maintain a $(1 + \epsilon)$ -approximation under edge insertions/deletions that runs in time $\mathcal{O}(\sqrt{m} \epsilon^{-2 - \mathcal{O}(1/\epsilon)} \log N)$ time per update, if edge weights are in between 1 and N . Their result is based on rerunning a static algorithm from time to time, a trimming routine that trims the graph to a smaller equivalent graph whenever possible and in the weighted case, a partition of the weights of the edges into intervals of geometrically increasing size. Stubbs and Williams [225] presented metatheorems for dynamic weighted matching. Here, the authors reduced the dynamic maximum weight matching problem to the dynamic maximum cardinality matching problem in which the graph is unweighted. The authors proved that using this reduction, if there is an α -approximation for maximum cardinality matching with update time T in an unweighted graph, then there is also a $(2 + \epsilon)\alpha$ -approximation for maximum weight matching with update time $\mathcal{O}(\frac{T}{\epsilon^2} \log^2 N)$. Their basic idea is an extension of the algorithm of Crouch and Stubbs [64] who tackled the problem in the streaming model. Here, the reduction is to take matchings from weight-threshold based subgraphs of the dynamic graph, i.e. the algorithm maintains maximal matchings in $\log C$ subgraphs, where subgraph i contains all edges having weight at least $(1 + \epsilon)^i$. The resulting matchings are then greedily merged together by considering the matched edges in descending order of i (heaviest edges first). Recently, the approach by Stubbs and Williams has been evaluated experimentally and has been compared against a new random walk-based approach [16] which gives a $(1 + \epsilon)$ approximation w.h.p.. When inserting or deleting an edge, the random walk-based approach finds random simple paths (using random walks) and solves those paths using dynamic programming to improve the maintained matching. In practice, the random walk-based approach outperforms the approach by Stubbs and Williams significantly.

3.5 k -Core Decomposition

A k -core of a graph is a maximal connected subgraph in which all vertices have degree at least k . The k -core decomposition problem is to compute the core number of every node in the graph. It is well-known that a k -core decomposition can be computed in linear time for a static graph. The problem of maintaining the k -core decomposition in a fully dynamic graph has not received much attention by the theoretical computer science community: Sun et al. [226] showed that the insertion and deletion of a single edge can change the core value of all vertices. They also gave a $(4 + \epsilon)$ -approximate fully dynamic algorithm with polylogarithmic running time. The algorithm can be implemented in time $\mathcal{O}(\log^2 n)$ in graphs using the algorithm of [45]. It dynamically maintains $\mathcal{O}(\log_{(1+\epsilon)} n)$ many (α, β) -decompositions of the graph, one for each β -value that is a power of $(1 + \epsilon)$ between 1 and $(1 + \epsilon)n$. An (α, β) -decomposition of a graph $G = (V, E)$ is a decomposition Z_1, \dots, Z_L of V into $L := 1 + \lceil (1 + \epsilon) \log n \rceil$ levels such that $Z_{i+1} \subseteq Z_i$ for all $1 \leq i < L$, $Z_1 = V$, and the following invariants are maintained: (1) All vertices v on level Z_i with $\deg_{Z_i}(v) > \alpha\beta$ belong to Z_{i+1} and (2) all vertices v on level Z_i with $\deg_{Z_i}(v) < \beta$ do not belong to Z_{i+1} . There are no further lower bounds, neither conditional nor unconditional, and no faster algorithms known for maintaining an approximate k -core decomposition.

Miorandi and De Pellegrini [169] proposed two methods to rank nodes according to their k -core number in fully dynamic networks. The focus of their work is to identify the most influential spreaders in complex dynamic networks. Li et al. [157] used a filtering method to only update nodes whose core number is affected by the network update. More precisely, the authors showed that nodes that need to be updated must be connected via a path to the endpoints of the inserted/removed edge and the core number must be equal to the smaller core number of the endpoints. Moreover, the authors presented efficient algorithms to identify such nodes as well as additional techniques to reduce the size of the nodes that need updates. Similarly, Sariyüce et al. [208] proposed the k -core algorithm TRAVERSAL and gave additional rules to prune the size of the subgraphs that are guaranteed to contain the vertices whose k -core number can have changed. Note that these algorithm can have a high variation in running time for the update operations depending on the size of the affected subgraphs. Zhang et al. [250] noted that due to this reason it can be impractical to process updates one by one and introduced the k -order concept which can reduce the cost of the update operations. A k -order is defined as follows: a node u is ordered before v in the k -order if u has a smaller core number than v or when the vertices have the same core number, if the linear time algorithm to compute the core decomposition would remove u before v . A recent result by Sun et al. [226] also contains experimental results. However, their main focus is on hypergraphs and there are no comparisons against the algorithms mentioned above.

Aridhi et al. [18] gave a distributed k -core decomposition algorithm in large dynamic graphs. The authors used a graph partitioning approach to distribute the workload and pruning techniques to find nodes that are affected by the changes. Wang et al. [242] gave a parallel algorithm that appears to significantly outperform the TRAVERSAL algorithm. Jin et al. [138] presented a parallel approach based on matching to update core numbers in fully dynamic networks. Specifically, the authors showed that if a batch of inserted/deleted edges forms a matching, then the core number update step can be performed in parallel. However, the type of the edges has to be the same (i.e. only insertions, or only deletions) in each update. Hua et al. [130] noted that previous algorithms become inefficient for high superior degree vertices, i.e. , vertices that have many neighbors that have a core number that is larger than its own core number. For example, the matching-based approach of Jin et al. [138] can only process one edge associated to a vertex in each iteration. Their new algorithm can handle multiple insertions/deletions per iteration.

It would be interesting to evaluate the algorithm of Sun et al. [226] which maintains a $(4 + \epsilon)$ -approximate core number, on graphs to see how far from the exact core numbers these estimates are and how its running time compares to the above approaches. Note that an (α, β) -decomposition actually gives a $(2\alpha + \epsilon)$ approximation and α has to be chosen to be slightly larger than 2 only to guarantee polylogarithmic updates. Thus, it would be interesting to also experiment with smaller values of α .

3.6 Motif Search and Motif Counting

Two graphs are isomorphic if there is a bijection between the vertex sets of the graphs that preserves adjacency. Given a graph pattern H (or multiple H_i), motif counting counts the subgraphs of G that are isomorphic to H (H_i respectively). In the work that is currently available there is a subset of work that focuses on the special case of counting triangles or wedges, i.e., paths of length two, in dynamic networks.

There is a conditional lower bound of $(\text{poly}, m^{1/2-\delta}, m^{1-\delta})$ even for the most fundamental problem of detecting whether a graph contains a triangle [117]. The same lower bound also extends to various four-vertex subgraphs [108], whereas there is a lower bound of

(poly, $m^{1-\delta}$, $m^{2-\delta}$) for counting 4-cliques as well as *induced* connected four-vertex subgraphs. A fully dynamic algorithm with $\mathcal{O}(\sqrt{m})$ update time was recently given independently by Kara et al. [142, 143] for counting triangles. Subsequently, Lu and Tao [161] studied the trade-off between update time and approximation quality and presented a new data structure for exact triangle counting whose complexity depends on the arboricity of the graph. The result by Kara et al. was also extended to general k -clique counting by Dhulipala et al. [74]. Motivated by the fact that real-world graphs in certain applications often have small h -index h (i.e., there are at most h vertices of degree at least h), Eppstein and Spiro [82] showed that the undirected triangle count can be maintained in $\mathcal{O}(h)$ time. Eppstein et al. [81] later extended this result to maintaining the counts of directed triangles in amortized $\mathcal{O}(h)$ time and of undirected four-vertex subgraphs in amortized $\mathcal{O}(h^2)$. Note that h can be as large as $\mathcal{O}(\sqrt{m})$, resulting in an amortized time complexity of $\mathcal{O}(m)$ per update for four-vertex subgraphs in general. Only very recently, Hanauer et al. [108] showed how to reduce this to amortized $\mathcal{O}(m^{2/3})$ time per update for all four-vertex subgraphs except the 4-clique. This is currently an active area of research.

In our description of the empirical work for this problem we start with recent work that mainly focuses on triangle counting. Pavan et al. [185] introduced neighborhood sampling to count and sample triangles in a one-pass streaming algorithm. In neighborhood sampling, first a random edge in the stream is sampled and in subsequent steps, edges that share an endpoint with the already sampled edges are sampled. The algorithm outperformed their implementations of the previous best algorithms for the problem, namely the algorithms by Jowhari and Ghodsi [140] and by Buriol et al. [54]. Note that the method does not appear to be able to handle edge deletions. Bulteau et al. [53] estimated the number of triangles in fully dynamic streamed graphs. Their method adapts 2-path sampling to work for dynamic graphs. The main idea of 2-path sampling is to sample a certain number of 2-paths and compute the ratio of 2-paths in the sample that are complete triangles. The total number of 2-paths in the graph is then multiplied with the ratio to obtain the total number of 2-paths in the graph. This approach fails, however, if one allows deletions. Thus, the contribution of the paper is a novel technique for sampling 2-paths. More precisely, the algorithm first streams the graph and sparsifies it. Afterwards, the sampling technique is applied on the sparsified graph. The core contribution of the authors is to show that the estimate obtained in the sparsified graph is similar to the number of triangles in the original graph. For graphs with constant transitivity coefficient, the authors achieve constant processing time per edge. Makkar et al. [163] presented an exact and parallel approach using an inclusion-exclusion formulation for triangle counting in dynamic graphs. The algorithm is implemented in `cuSTINGER` [84] and runs on GPUs. The algorithm computes updates for batches of edge updates and also updates the number of triangles each vertex belongs to. The `TRIËST` algorithm [224] estimates local and global triangles. An input parameter of the algorithm is the amount of available memory. The algorithm maintains a sample of the edges using reservoir sampling and random pairing to exploit the available memory as much as possible. The algorithm reduces the average estimation error by up to 90% w.r.t. to the previous state-of-the-art. Han and Sethu [107] proposed a new sampling approach, called `edge-sample-and-discard`, which generates an unbiased estimate of the total number of triangles in a fully dynamic graph. The algorithm significantly reduces the estimation error compared to `TRIËST`. The `MASCOT` algorithm [159, 158] focuses on local triangle counting, i.e. counting the triangles adjacent to every node. In their work, the authors provide an unbiased estimation of the number of local triangles.

We now report algorithms that can count more complex patterns. The neighborhood sampling method of Pavan et al. [185] can also be used for more complex patterns, for example Pavan et al. also presented experiments for 4-cliques. Shiller et al. [212] presented the

stream-based (insertions and deletions) algorithm **StreaM** for counting undirected 4-vertex motifs in dynamic graphs. Ahmed et al. [8] presented a general purpose sampling framework for graph streams. The authors proposed a martingale formulation for subgraph count estimation and showed how to compute unbiased estimate of subgraph counts from a sample at any point during the stream. The estimates for triangle and wedge counting obtained are less than 1% away from the true number of triangles/wedges. The algorithm outperformed their own implementation of **TRIÈST** and **MASCOT**. Mukherjee et al. [171] gave an exact counting algorithm for a given set of motifs in dynamic networks. Their focus is on biological networks. The algorithm computes an initial embedding of each motif in the initial network. Then for each motif its embeddings are stored in a list. This list is then dynamically updated while the graph evolves. Liu et al. [160] estimated motifs in dynamic networks. The algorithm uses exact counting algorithms as a subroutine, and hence can speed up any exact algorithm at the expense of accuracy. The main idea of their algorithm is to partition the stream into time intervals and find exact motif counts in subsets of these intervals. Recently, Wang et al. [241] improved on the result of Liu et al.. The improvement stems from a generic edge sampling algorithm to estimate the number of instances of any k -vertex ℓ -edge motif in a dynamic network. The main idea of the algorithm is to first uniformly at random draw random edges from the dynamic network, then exactly count the number of local motifs and lastly estimate the global count from the local counts. The experimental evaluation showed that their algorithm is up to 48.5 times faster than the previous state-of-the-art while having lower estimation errors.

Dhulipala et al. [74] recently gave parallel batch-dynamic algorithms for k -clique counting. Their first algorithm is a batch-dynamic parallel algorithm for triangle counting that has amortized work $\mathcal{O}(\Delta\sqrt{\Delta+m})$ and $\mathcal{O}(\log^*(\Delta+m))$ depth with high probability. The algorithm is based on degree thresholding which divides the vertices into vertices with low- and high-degree. Given the classification of the vertex, different updates routines are used. A multicore implementation of the triangle counting algorithm is given. Experiments indicate that the algorithms achieve 36.54 to 74.73-times parallel speedups on a machine with 72 cores. Lastly, the authors developed a simple batch-dynamic algorithm for k -clique counting that has expected $\mathcal{O}(\Delta(m+\Delta)\alpha^{k-4})$ work and $\mathcal{O}(\log^{k-2}n)$ depth with high probability, for graphs with arboricity α .

To summarize for this problem the empirical work is far ahead of the theoretical work and it would be interesting to better understand the theoretical complexity of motif search and motif counting.

3.7 Diameter

The *eccentricity* of a vertex is the greatest distance between the vertex and any other vertex in the graph. Based on this definition, the *diameter* of a graph is defined as the maximum eccentricity over all vertices in the graph. The *radius* is the minimum eccentricity of all vertices. Through recomputation from scratch it is straightforward to compute a 2-approximation for diameter and radius and a $(2+\epsilon)$ -approximation for radius in linear time.

Anacona et al. [15] recently showed that under the strong exponential time hypothesis (SETH) there can be no $(2-\epsilon)$ -approximate fully dynamic approximation algorithm for any of these problems with $\mathcal{O}(m^{1-\delta})$ update or query time for any $\delta > 0$. There also exist non-trivial (and sub- n^2 time) fully dynamic algorithms for $(1.5+\epsilon)$ approximate diameter (and also for radius and eccentricities) [234]. In this paper, the authors also construct a non-trivial algorithm for exact diameter. We are not aware of any experimental study for fully dynamic diameter.

3.8 Independent Set and Vertex Cover

Given a graph $G = (V, E)$, an *independent set* is a set $S \subseteq V$ such that no vertices in S are adjacent to one another. The *maximum independent set problem* is to compute an independent set of maximum cardinality, called a *maximum independent set* (MIS). The *minimum vertex cover* problem is equivalent to the maximum independent set problem: S is a minimum vertex cover C in G iff $V \setminus S$ is a maximum independent set $V \setminus C$ in G . Thus, an algorithm that solves one of these problems can be used to solve the other. Note, however, that this does not hold for approximation algorithms: If C' is an α -approximation of a minimum vertex cover, then $V \setminus C'$ is not necessarily an α -approximation of a maximum independent set. Another related problem is the *maximal independent set* problem. A set S is a maximal independent set if it is an independent set such that for any vertex $v \in V \setminus S$, $S \cup \{v\}$ is not independent.

As computing the size of an MIS is NP-hard, all dynamic algorithms of independent set study the maximal independent set problem. Note, however, that unlike for matching a maximal independent set does not give an approximate solution for the MIS problem, as shown by a star graph. In a sequence of papers [19, 103, 20, 60, 31] the running time for the maximal independent set problem was reduced to $\mathcal{O}(\log^4 n)$ expected worst-case update time.

While quite a large amount of engineering work has been devoted to the computation of independent sets/vertex covers in static graphs, the amount of engineering work for the dynamic independent set problem is very limited. Zheng et al. [252] presented a heuristic fully dynamic algorithm and proposed a lazy search algorithm to improve the size of the maintained independent set. A year later, Zheng et al. [251] improved the result such that the algorithm is less sensitive to the quality of the initial solution used for the evolving MIS. In their algorithm, the authors used two well known data reduction rules, degree one and degree two vertex reduction, that are frequently used in the static case. Moreover, the authors can handle batch updates. Bhore et al. [48] focused on the special case of MIS for independent rectangles which is frequently used in map labelling applications. The authors presented a deterministic algorithm for maintaining a MIS of a dynamic set of uniform rectangles with amortized sub-logarithmic update time. Moreover, the authors evaluated their approach using extensive experiments.

3.9 Shortest Paths

One of the most studied problems on weighted dynamic networks is the maintenance of shortest path information between pairs of vertices. In the most general setting, given an undirected, dynamic graph with dynamically changing edge weights representing distances, we are interested in the shortest path between two arbitrary vertices s and t (*all-pairs shortest path problem*). For the *single-source shortest path problem*, the source vertex s is fixed beforehand and the dynamic graph algorithm is only required to answer distance queries between s and an (arbitrary) vertex t which is specified by the query operation. In the *s-t shortest path problem* both s and t are fixed beforehand and the data structure is only required to return the distance between s and t as answer to a query. In all cases, the analogous problem can also be cast on a directed graph, asking for a shortest path from s to t instead.

Let $\delta > 0$ be a small constant. There is a conditional lower bound of $(\text{poly}, m^{1/2-\delta}, m^{1-\delta})$ for any small constant $\delta > 0$ based on the OMv conjecture, even for s - t shortest paths [117]. This lower bound applies also to any algorithm that gives a better than $5/3$ -approximation. For planar graphs the product of query and update time is $\Omega(n^{1-\delta})$ based on the APSP

conjecture [2]. As even the partially dynamic versions have shown to be at least as hard as the static all-pairs shortest paths problem [199, 2], one cannot hope for a *combinatorial* fully dynamic all-pairs shortest paths algorithm with $\mathcal{O}(n^{3-\delta})$ preprocessing time, $\mathcal{O}(n^{2-\delta})$ amortized update time, and constant query time. The state-of-the-art algorithms come close to this: For directed, weighted graphs, Demetrescu and Italiano [72] achieved an *amortized* update time of $\tilde{\mathcal{O}}(n^2)$, which was later improved by a polylogarithmic factor by Thorup [228]. Both of these algorithms actually allow vertex insertions and deletion, not just edge updates. There is also a fully dynamic $2^{O(k^2)}$ -approximation algorithm that takes time $\tilde{\mathcal{O}}(\sqrt{mn}^{1/k})$ per update and $O(k^2)$ per update for any positive integer k [5].

With respect to *worst-case* update times, the currently fastest algorithms are randomized with $\tilde{\mathcal{O}}(n^{2+2/3})$ update time [4, 106]. Moreover, Probst Gutenberg and Wulff-Nilsen [106] presented a deterministic algorithm with $\tilde{\mathcal{O}}(n^{2+5/7})$ update time, thereby improving a 15 years old result by Thorup [229]. Van den Brand and Nanongkai [233] showed that Monte Carlo-randomized $(1+\epsilon)$ -approximation algorithms exist with $\tilde{\mathcal{O}}(n^{1.823}/\epsilon^2)$ worst-case update time for the fully dynamic single-source shortest path problem and $\tilde{\mathcal{O}}(n^{2.045}/\epsilon^2)$ for all-pairs shortest paths, in each case with positive real edge weights and constant query time. Slightly faster exact and approximative algorithms exist in part for the “special cases” of *unweighted* graphs [206, 199, 4, 106, 234, 233] (all edges have unit weight) and/or *undirected* graphs [200, 233] (every edge has a reverse edge of the same weight). More details on shortest paths algorithms including fully dynamic algorithms are given in the survey of Madkour et al. [162].

The first experimental study for fully dynamic single-source shortest paths on directed graphs with positive real edge weights was conducted by Frigioni et al. [87], who evaluated Dijkstra’s seminal static algorithm [75] against a fully dynamic algorithm by Ramalingam and Reps [195] (RR) as well as one by Frigioni et al. [88] (FMN). RR is based on Dijkstra’s static algorithm and maintains a spanning subgraph consisting of edges that belong to at least one shortest s - t path for some vertex t . After an edge insertion, the spanning subgraph is updated starting from the edge’s head until all affected vertices have been processed. In case of an edge deletion, the affected vertices are identified as a first step, followed by an update of their distances. The resulting worst-case update time is $\mathcal{O}(x_\delta + n_\delta \log n_\delta) \subseteq \mathcal{O}(m + n \log n)$, where n_δ corresponds to the number of vertices affected by the update, i.e., whose distance from s changes and x_δ equals n_δ plus the number of edges incident to an affected vertex. Similarly, Frigioni et al. [88] analyzed the update complexity of their algorithm FMN with respect to the change in the solution and showed a worst-case running time of $\mathcal{O}(|U_\delta| \sqrt{m} \log n)$, where U_δ is the set of vertices where either the distance from s must be updated or their parent in the shortest paths tree. The algorithm assigns each edge (u, v) a forward (backward) level, which corresponds to the difference between the (sum of) v ’s (u ’s) distance from s and the edge weight, as well as an owner, which is either u or v , and used to bound the running time. Incident outgoing and incoming edges of a vertex that it does not own are kept in a priority queue each, with the priority corresponding to the edge’s level. In case of a distance update at a vertex, only those edges are scanned that are either owned by the vertex or have a priority that indicates a shorter path. Edge insertion and deletion routines are based on Dijkstra’s algorithm and handled similar as in RR, but using level and ownership information. The experiments were run on three types of input instances: randomly generated ones, instances crafted specifically for the tested algorithms, and random updates on autonomous systems networks. The static Dijkstra algorithm is made dynamic in that it is re-run from scratch each time its shortest paths tree is affected by an update. The evaluation showed that the dynamic algorithms can speed up the update time by 95 %

over the static algorithm. Furthermore, RR turned out to be faster in practice than FMN except on autonomous systems instances, where the spanning subgraph was large due to many alternative shortest paths. In a follow-up work, Demetrescu et al. [71, 70] extended this study to dynamic graphs with arbitrary edge weight, allowing in particular also for negative weights. In addition to the above mentioned algorithm by Ramalingam and Reps [195] in a slightly lighter version (RRL) and the one by Frigioni et al. [88] (FMN), their study also includes a simplified variant of the latter which waives edge ownership (DFMN), as well as a rather straightforward dynamic algorithm (DF) that in case of a weight increase on an edge (u, v) first marks all vertices in the shortest paths subtree rooted at v and then finds for each of these vertices an alternative path from s using only unmarked vertices. The new weight of these vertices can be at most this distance or the old distance plus the amount of weight increase on (u, v) . Therefore, the minimum is taken as a distance estimate for the second step, where the procedure is as in Dijkstra's algorithm. In case of a weight decrease on an edge (u, v) the first step is omitted. As Dijkstra's algorithm is employed as a subroutine, the worst-case running time of DF for a weight change is $\mathcal{O}(m + n \log n)$. For updates, all algorithms use a technique introduced by Edmonds and Karp [78] to transform the weight $w(u, v)$ of each edge (u, v) to a non-negative one by replacing it with the reduced weight $w(u, v) - (d(v) - d(u))$, where $d(\cdot)$ denotes the distance from s . This preserves shortest paths and allows Dijkstra's algorithm to be used during the update process. The authors compared these dynamic algorithms to re-running the static algorithm by Bellman and Ford on each update from scratch on various randomly generated dynamic instances with mixed incremental and decremental updates on the edge weights, always avoiding negative-length cycles. Their study showed that DF is the fastest in practice on most instances, however, in certain circumstances RR and DFMN are faster, whereas FMN turned out to be too slow in practice due to its complicated data structures. The authors observed a runtime dependency on the interval size of the edge weights; RR was the fastest if this interval was small, except for very sparse graphs. DFMN on the other hand was shown to perform better than DF in presence of zero-length cycles, whereas RR is incapable of handling such instances. It is interesting to note here that the differences in running time are only due to the updates that increase distances, as all three candidates used the same routine for operations that decrease distances. The static algorithm was slower than the dynamic algorithms by several orders of magnitude.

The first fully dynamic algorithm for all-pairs shortest paths in graphs with positive integer weights less than a constant C was presented by King [147], with an amortized update time of $\mathcal{O}(n^{2.5} \sqrt{C \log n})$. For each vertex v , it maintains two shortest paths trees up to a distance d : one outbound with v as source and one inbound with v as target. A so-called stitching algorithm is used to stitch together longer paths from shortest paths of distance at most d . To achieve the above mentioned running time, d is set to $\sqrt{nC \log n}$. The space requirement is $\mathcal{O}(n^3)$ originally, but can be reduced to $\tilde{\mathcal{O}}(n^2 \sqrt{nC})$ [148]. For non-negative, real-valued edge weights, Demetrescu and Italiano [72] proposed an algorithm with an amortized update time of $\mathcal{O}(n^2 \log^3 n)$, which was later improved to $\mathcal{O}(n^2 (\log n + \log^2((n+m)/n)))$ by Thorup [228]. The algorithm uses the concept of *locally shortest paths*, which are paths such that each proper subpath is a shortest path, but not necessarily the entire path, and *historical shortest paths*, which are paths that have once been shortest paths and whose edges have not received any weight updates since then. The combination of both yields so-called locally historical paths, which are maintained by the algorithm. To keep their number small, the original sequence of updates is transformed into an equivalent, but slightly longer *smoothed sequence*. In case of a weight update, the algorithm discards all maintained paths containing the updated

edge and then computes new locally historical paths using a routine similar to Dijkstra's algorithm. Both algorithms have constant query time and were evaluated experimentally in a study by Demetrescu and Italiano [73] against RRL [71] on random instances, graphs with a single bottleneck edge, which serves as a bridge between two equally-sized complete bipartite graphs and only its weight is subject to updates, as well as real-world instances obtained from the US road networks and autonomous systems networks. Apart from RRL, the study also comprises the Dijkstra's static algorithm. Both these algorithms are designed for single-source shortest paths and were hence run once per vertex. All algorithms were implemented with small deviations from their respective theoretical description to speed them up in practice. The study showed that RRL and the algorithm based on locally historical paths (LHP) can outperform the static algorithm by a factor of up to 10 000, whereas the algorithm by King only achieves a speedup factor of around 10. RRL turned out to be especially fast if the solution changes only slightly, but by contrast exhibited the worst performance on the bottleneck instances unless the graphs were sparse. In comparison, LHP was slightly slower on sparse instances, but could beat RRL as the density increased. The authors also point out differences in performance that depend mainly on the memory architecture of the machines used for benchmarking, where RRL could better cope with small caches or memory bandwidth due to its reduced space requirements and better locality in the memory access pattern, whereas LHP benefited from larger caches and more bandwidth.

Buriol et al. [55] presented a technique that reduces the number of elements that need to be processed in a heap after an update for various dynamic shortest paths algorithms by excluding vertices whose distance changes by exactly the same amount as the weight change and handling them separately. They showed how this improvement can be incorporated into RR [195], a variant similar to RRL [195], the algorithm by King and Thorup [148] (KT), and DF [70] and achieves speedups of up to 1.79 for random weight changes and up to 5.11 for unit weight changes. Narváez et al. [173] proposed a framework to dynamize static shortest path algorithms such as Dijkstra's or Bellman-Ford [32]. In a follow-up work [174], they developed a new algorithm that fits in this framework and is based on the linear programming formulation of shortest paths and its dual, which yields the problem in a so-called *ball-and-string model*. The authors experimentally showed that their algorithm needs fewer comparisons per vertex when processing an update than the algorithms from their earlier work, as it can reuse intact substructures of the old shortest path tree.

To speed up shortest paths computations experimentally, Wagner et al. [239] introduced a concept for pruning the search space by *geometric containers*. Here, each edge (u, v) is associated with a set of vertices called container, which is a superset of all vertices w whose shortest u - w path starts with (u, v) . The authors assume that each vertex is mapped to a point in two-dimensional Euclidean space and based on this, suggest different types of geometric objects as containers, such as disks, ellipses, sectors or boxes. All types of container only require constant additional space per edge. The experimental evaluation on static instances obtained from road and railway networks showed that using the bounding box as container reduces the query time the most in comparison to running the Dijkstra algorithm without pruning, as the search space could be reduced to 5% to 10%. This could be preserved for dynamic instances obtained from railway networks if containers were grown and shrunk in response to an update, with a speedup factor of 2 to 3 over a recomputation of the containers from scratch. For bidirectional search, reverse containers need to be maintained additionally, which about doubled the absolute update time. Dellinger and Wagner [69] adapted the static ALT algorithm [90] to the dynamic setting. ALT is a variant of bidirectional A^* search that uses a small subset of vertices called *landmarks*, for which distances from and to all other

vertices are precomputed, and the triangle inequality to direct the search for a shortest path towards the target more efficiently. The authors distinguish between an eager and a lazy dynamic version of ALT, where the eager one updates all shortest path trees of the landmarks immediately after an update. The lazy variant instead keeps the preprocessed information as long as it still guarantees correctness, which holds as long as the weight of an edge is at least its initial weight, however at the expense of a potentially larger search space. The choice of landmarks remains fixed. The experimental study on large road networks showed that queries in the lazy version are almost as fast as in the eager version for short distances or if no edges representing motorways are affected, but slower by several factors for longer distances, larger changes to the weight of motorway edges, or after many updates. Schultes and Sanders [216] combined and generalized different techniques that have been successfully used in the static setting, such as separators, highway hierarchies, and transit node routing in a multi-level approach termed *highway-node routing*: For the set of vertices V_i on each level i , $V_i \subseteq V_{i-1}$, and the overlay graph G_i is defined on V_i with an edge $(s, t) \in V_i \times V_i$ iff there is a shortest s - t path in G_{i-1} that contains no vertices in V_i except for s and t . Queries are carried out by a modified Dijkstra search on this graph hierarchy. The authors extended this approach also to the dynamic setting and consider two scenarios: a server scenario, where in case of edge weight changes the sets of highway nodes V_i are kept and the graphs G_i are updated, and a mobile scenario, where only those vertices that are potentially affected are determined and the query routine needs to be aware of possibly outdated information during a search. In an experimental evaluation on a very large road network with dynamically changing travel times as weights it is shown that the dynamic highway-node routing outperformed recomputation from scratch as well as dynamic ALT search with 16 landmarks clearly with respect to preprocessing, update, and query time as well as space overhead.

Misra and Oommen [170] presented algorithms for single-source shortest paths that are based on learning automata and designed to find “statistical” shortest paths in a stochastic graph with stochastically changing edge weights. The algorithms are extensions of RR [195] and FMN [88] and shown to be superior to the original versions of RR and FMN by several orders of magnitude once they have converged. Chan and Yang [58] studied the problem of dynamically updating a single-source shortest path tree under multiple concurrent edge weight updates. They amended the algorithm by Narváez et al. [174] (MBS), for which they showed that it may misbehave in certain circumstances and suggested two further algorithms: MFP is an optimized version of an algorithm by Ramalingam and Reps [194] (DynamicSWSF-FP), which can handle multiple updates at once. The second algorithm is a generalization of the dynamic Dijkstra algorithm proposed by Narváez et al. [173]. In a detailed evaluation, they showed that an algorithm obtained by combining the incremental phase of MBS and the decremental phase of their dynamization of Dijkstra’s algorithm performed best on road networks, whereas the dynamized Dijkstra’s algorithm was best on random networks. An extensive experimental study on single-source shortest path algorithms was conducted by Bauer and Wagner [29]. They suggested several tuned variants of DynamicSWSF-FP [194] and evaluated them against FMN [88], different algorithms from the framework by Narváez et al. [173], as well as RR [195] on a diverse set of instances. The algorithms from the Narváez framework showed similar performance in case of single-edge updates and were the fastest on road networks and generated grid-like graphs. By contrast, the tuned variants of DynamicSWSF-FP behaved less consistent. RR was superior on Internet networks, whereas FMN was the slowest, especially on sparse instances. Interestingly, the authors showed that for batch updates with a set of randomly chosen edges, the algorithms behave similar as for single-edge updates, as there was almost no interference. The picture changed slightly for

simulated node failures and strongly for simulated traffic jams. RR and a tuned variant of DynamicSWSF-FP showed the best performance for simulated node failures, and two tuned variants of DynamicSWSF-FP dominated in case of simulated traffic jams. Notably, the algorithms from the Narváez framework were faster here if instead of in batches, the updates were processed one-by-one. In follow-up works, D’Andrea et al. [65] evaluated several batch-dynamic algorithms for single-source shortest paths, where the batches are homogeneous, i.e., all updates are either incremental or decremental. Their study contains RR [195], a tuned variant of DynamicSWSF-FP [194] described by Bauer and Wagner [29] (TSWSF), as well as a new algorithm DDFLP, which is designed specifically to handle homogeneous batches and uses similar techniques as FMN [88]. The instance set comprised road and Internet networks as well as randomly generated graphs according to the Erdős-Rényi model (uniform degree distribution) and the Barabási-Albert model (power-law degree distribution). Batch updates were obtained from simulated node failure and recovery, simulated traffic jam and recovery, as well as randomly selected edges for which the weights were either increased or decreased randomly. The evaluation confirmed the results by Bauer and Wagner [29] and showed that DDFLP and TSWSF are best in case of update scenarios like node failures or traffic jams and otherwise TSWSF and RR, where RR is preferable to TSWSF if the interference among the updates is low and vice versa. DDFLP generally benefited from dense instances. Singh and Khare [221] presented the first batch-dynamic parallel algorithm for single-source shortest paths for GPUs and showed in experiments that it outperforms the (sequential) tuned DynamicSWSF-FP algorithm [29] by a factor of up to 20 if the distances of up to 10% of the nodes are affected.

For real-time shortest path computations on networks with fixed topology, but varying metric, Dellling et al. [68] suggested a three-stage approach: In the first, preprocessing step, a metric-independent, moderate amount of auxiliary data is obtained from the network’s topology. It is followed by a customization step, which is run for each metric and produces few additional data. Whereas the first phase is run only once and can therefore use more computation time, the second phase must complete within seconds in real-life scenarios. Shortest path queries form the third phase and must be fast enough for actual applications. For the first, metric-independent stage, the authors describe an approach based on graph partitioning, where the number of *boundary edges*, i.e., edges between different partitions, is to be minimized. For the second stage, they compute an overlay network consisting of shortest paths between all pairs of *boundary nodes*, i.e. nodes that are incident to at least one boundary edge. An s - t query is then answered by running a bidirectional Dijkstra algorithm on the graph obtained by combining the overlay graph with the subgraphs induced by the partitions containing s and t , respectively. The authors also considered various options for speedups, such as a sparsification of the overlay network, incorporating goal-directed search techniques, and multiple levels of overlays. An experimental evaluation on road networks with travel distances and travel times as metrics showed that their approach allows for real-time queries and needs only few seconds for the metric-dependent customization phase.

Arc flags belong in the category of goal-directed techniques to speed up shortest path computations and have been successfully used in the static setting [28]. To this end, the set of vertices is partitioned into a number of regions. Each edge receives a label consisting of a flag for each region, which tells whether there is a shortest path starting with this edge and ending in the region. The technique is related to geometric containers and uses the arc flags to prune a (bidirectional) Dijkstra search. Berettini et al. [38] were the first to consider arc flags in a dynamic setting, however only for the case of weight increases. Their main idea is to maintain a threshold for each edge and region that gives the increase in weight

required for the edge to lie on a shortest path. On a weight increase, the thresholds are updated and used to determine when to change an arc flag. Although this potentially reduces the quality of the arc flags with each update, the experimental evaluation showed that the increase in query time is very small as long as the update sequence is short. With respect to the update time, a significant speedup could be achieved over recomputing arc flags from scratch. To refresh arc flags more exactly and in a fully dynamical setting, D'Angelo et al. [66] introduced a data structure called *road signs*. Road signs complement arc flags and store for each edge e and region R the set of boundary nodes contained in any shortest path starting with e and ending in R . In case of a weight increase, the algorithm first identifies all affected nodes whose shortest path to a boundary node changed and then updates all road signs for all outgoing edges of an affected node. In case of a weight decrease on edge (u, v) , the authors observed that all shortest paths containing (u, v) remain unchanged. However, shortest paths starting with other outgoing or incoming edges of u might require updates, as well as other paths containing an incoming edge of u . In an experimental study on road networks, the authors compared their algorithm against one that recomputes arc flags from scratch as well as the algorithm by Berettini et al. [38] (BDD). To mimic traffic jams and similar occurrences, the weight of a randomly chosen edge increases and then decreases by the same amount, however not necessarily in subsequent updates. The evaluation showed that updating both road signs and arc flags is by several factors faster than recomputing arc flags from scratch. On instances with weight increases only, the authors showed that their new algorithm outperforms BDD distinctly both for updates and queries.

A further speedup technique for shortest path queries are *2-hop cover labelings*, where the label $L(v)$ of each node v is a carefully chosen set of nodes U_v along with the distance between v and u for each $u \in U$. For each pair of vertices s and t , the shortest s - t path can be obtained by intersecting U_s and U_t and taking the minimum over all combinations of s - x and x - t paths for all nodes $x \in U_s \cap U_t$. In the static setting, a 2-hop cover labeling can be computed based on a breadth-first search that is run once for every vertex (“naive landmark labeling”). Akiba et al. [9] introduced *pruned landmark labeling* (PLL), which constitutes a more refined approach and uses pruned breadth-first searches instead. The authors developed an incremental algorithm for PLL, which was complemented by D'Angelo et al. [67] to a fully dynamic algorithm. The experimental evaluation showed that the algorithm achieves speedups of several orders of magnitude over a recomputation from scratch, while at the same time preserving the quality of the labeling, which makes this speedup technique suitable for practical use in dynamic scenarios.

Hayashi et al. [111] proposed a method to support shortest paths queries on unweighted networks with billions of edges by combining a bidirectional breadth-first search, which is optimized for the structure of small-world networks, with landmarks. To this end, the authors choose high-degree vertices and store shortest path trees as well as those of a subset of their neighbors in a so-called “bit-parallel” form. This increases the number of landmarks, which in turn generally speeds up the search and in particular for high-degree vertices, and at the same time keeps the memory requirements comparatively small. After an edge insertion or deletion, the bit-parallel shortest paths trees are updated accordingly. The experimental evaluation on twelve real-world instances having between 1.5 million and 3.7 billion edges showed that the new algorithm was able to process queries on average in less than 8 ms and even considerably less on many instances. The average edge insertion and deletion times were less than 1.3 ms and 8.1 s, respectively, after an initialization time of less than 1 h. The incremental algorithm by Akiba et al. [9], which was included in the study, was distinctly faster on queries, but on some instances several factors slower on insertions. However, it failed to complete the preprocessing step within 10 h or required more than 128 GB of memory on half of all instances.

3.10 Maximum Flows and Minimum Cuts

An instance of the maximum flow/minimum cut problem consists of an edge-weighted directed graph $G = (V, E, c)$ along with two distinguished vertices s and t . The edge weights c are positive and commonly referred to as *capacities*. An $(s-t)$ flow f is a non-negative weight function on the edges such $f(e) \leq c(e)$ for all $e \in E$ (capacity constraints) and except for s and t , the total flow on the incoming edges of each vertex must equal the total flow on the outgoing edges (conservation constraints). The *excess* of a vertex v is the total flow on its incoming edges minus that on its outgoing edges, which must be zero for all vertices except s and t . The value of a flow f then is the excess of t . The task is to find a flow of maximum value. An $(s-t)$ cut is a subset of edges $C \subseteq E$ whose removal makes t unreachable from s , and its value is the sum of the capacities of all edges in the cut. The well-known max-flow min-cut theorem states that the maximum value of a flow equals the minimum value of a cut. The fastest static algorithm whose running time does not depend on the size of the largest edge weight computes an optimal solution in $\mathcal{O}(nm)$ time [182].

In the dynamic setting, there is a conditional lower bound of $(\text{poly}, m^{1/2-\delta}, m^{1-\delta})$ (for any small constant $\delta > 0$) for the size of the maximum $s-t$ flow even in unweighted, undirected graphs based on the OMv conjecture [117]. Recently Chen et al. [62] gave an $\mathcal{O}(\log n \log \log n)$ -approximate fully dynamic maximum flow algorithm in time $\tilde{\mathcal{O}}(n^{2/3+o(1)})$ per update and Goranci et al. [95] gave a $n^{o(1)}$ -approximate fully dynamic algorithm in time $n^{o(1)}$ worst-case update time and $\mathcal{O}(\log^{1/6} n)$ query time. In the unweighted setting Jin and Sun [139] gave a datastructure that can be constructed for any fixed positive integer $c = (\log n)^{o(1)}$ and that answers for any pair (s, t) of vertices that are parameters of the query in time $n^{o(1)}$ where s and t are c -edge connected.

For *global minimum cuts* in the unweighted setting Thorup and Karger [231] presented a $\sqrt{2 + o(1)}$ -approximation algorithm that takes polylogarithmic time per update and query and Thorup [230] designed a $(1 + \epsilon)$ -approximate algorithm in $\tilde{\mathcal{O}}(\sqrt{n})$ time per update and query.

Kumar and Gupta [154] extended the preflow-push approach [93] to solve maximum flow in static graphs to the dynamic setting. A preflow is a flow under a relaxed conservation constraint in that the excess of all vertices except s must be non-negative. Vertices with positive excess are called *active*. Preflow-push algorithms, also called push-relabel algorithms, use this relaxed variant of a flow during the construction of a maximum flow along with distance labels on the vertices. Generally speaking, they push flow out of active vertices towards vertices with smaller distance (to t) and terminate with a valid flow (i.e., observing conservation constraints). In case of an edge insertion or deletion, Kumar and Gupta first identify affected vertices via forward and backward breadth-first search while observing and updating distance labels and then follow the scheme of a basic preflow-push algorithm, however restricted to the set of affected vertices. The authors evaluated their algorithm only for the incremental setting on a set of randomly generated instances against the static preflow-push algorithm in [93] and found that their algorithm is able to reduce the number of push and relabel operations significantly as long as the instances are sparse and the number of affected vertices remains small.

Many important fields of application for the maximum flow/minimum cut problem stem from computer vision. In this area, the static algorithm of Boykov and Kolmogorov [50] (BK) is widely used due to its good performance in practice on computer vision instances and despite its pseudo-polynomial worst-case running time of $\mathcal{O}(nm \cdot \text{OPT})$, with OPT being the value of a maximum flow/minimum cut. Interestingly, however, a study by Verma and Batra [238] shows that its practical superiority only holds for sparse instances. BK follows

the Ford-Fulkerson method of augmenting flow along s - t paths, but uses two search trees grown from s and t , respectively, to find such paths. Kohli and Torr [149, 150] extended BK to the fully dynamic setting by updating capacities and flow upon changes and discuss an optimization that tries to recycle the search trees. They experimentally compared their algorithm to repeated executions of the static algorithm on dynamic instances obtained from video sequences and achieve a substantial speedup. They also observed that reusing the search trees leads to longer s - t paths, which affects the update time negatively as the instances undergo many changes.

Goldberg et al. [91] developed EIBFS, a generalization of their earlier algorithm IBFS, that by contrast also extends to the dynamic setting in a straightforward manner. IBFS in turn is a modification of BK that ensures that the two trees grown from s and t are height-minimal (i.e., BFS trees) and is closely related to the concept of blocking flows. The running time of EIBFS in the static setting and thus the initialization in the dynamic setting, is $\mathcal{O}(mn \log(n^2/m))$ with dynamic trees or $\mathcal{O}(mn^2)$ without. The algorithm works with a so-called pseudoflow, which observes capacity constraints, but may violate conservation constraints. It maintains two vertex-disjoint forests S and T , where the roots are exactly those vertices with a surplus of incoming flow and those with a surplus of outgoing flow, respectively, and originally only contain s and t . The steps of the algorithm consist in growth steps, where S or T are grown level-wise, augmentation steps, which occur if a link between the forests has been established and flow is pushed to a vertex in the other forest and further on to the root, and adoption steps, where vertices in T with surplus incoming flow or vertices in S with surplus outgoing flow are either adopted by a new parent in the same forest or become a root in the other forest. In case of an update in the dynamic setting, the invariants of the forests are restored and flow is pushed where possible, followed by alternating augmentation and adoption steps if necessary. The authors also mention that resetting the forests every $\mathcal{O}(m)$ work such that they contain only vertices with a surplus outgoing or incoming flow seemed to be beneficial in practice. In their experimental evaluation of EIBFS against the algorithm by Kohli and Torr as well as an altered version thereof and a more naive dynamization of IBFS, they showed for different dynamic real-world instances from the field of computer vision that EIBFS is the fastest on eight out of fourteen instances and relatively robust: In contrast to its competitors, it always takes at most roughly twice the time of the fastest algorithm on an instance. Notably, no algorithm is dominated by another one across all instances.

Zhu et al. [253] described a dynamic update strategy based on augmenting and de-augmenting paths as well as cancelling cyclic flows. The latter serves as a preparatory step and only reroutes flow in the network without increasing or decreasing the total s - t flow and is only necessary in a decremental update operation. They experimentally evaluated the effectiveness of their algorithm for online semi-supervised learning, where real-world big data is classified via minimum cuts, and showed that their algorithm outperforms state-of-the-art stream classification algorithms. A very similar algorithm was proposed by Greco et al. [101]. The authors compared it experimentally against EIBFS and the dynamic BK algorithm by Kohli and Torr as well as a number of the currently fastest static algorithms. Their experiments were conducted on a set of instances from computer vision where equally many edges are randomly chosen to be inserted and deleted, respectively. They showed that their algorithm is with one exception always the fastest on average in performing edge insertions if compared to the average update time of the competitors, and on half of all instances also in case of edge deletions. On the remaining instances, the average update time of EIBFS dominated.

For the global minimum cut problem, Henzinger et al. [118] implemented an algorithm for large dynamic graphs under both edge insertions and deletions. For edge insertions, the algorithm uses the approach of Henzinger [120] and Goranci et al. [94], which maintain a compact data structure of all minimum cuts in a graph and invalidate only the minimum cuts that are affected by an edge insertion. For edge deletions, the algorithms use the push-relabel algorithm of Goldberg and Tarjan [92] to certify whether the previous minimum cut is still a minimum cut. The algorithm outperformed static approaches by up to five orders of magnitude on large graphs.

3.11 Graph Clustering

Graph clustering is the problem of detecting tightly connected regions of a graph. More precisely, a clustering \mathcal{C} is a partition of the set of vertices, i.e. a set of disjoint *clusters* of vertices V_1, \dots, V_k such that $V_1 \cup \dots \cup V_k = V$. However, k is usually not given in advance and some objective function that models intra-cluster density versus inter-cluster sparsity, is optimized. It is common knowledge that there is neither a single best strategy nor objective function for graph clustering, which justifies a plethora of existing approaches. Moreover, most quality indices for graph clusterings have turned out to be NP-hard to optimize and are rather resilient to effective approximations, see, e.g. [21, 51], allowing only heuristic approaches for optimization. There has been a wide-range of algorithms for static graph clustering, the majority are based on the paradigm of intra-cluster density versus inter-cluster sparsity. For dynamic graphs, there has been a recent survey on the topic of community detection [202]. The survey covers features and challenges of dynamic community detection and classifies published approaches. Here we focus on engineering results and extend their survey in that regard with additional references as well as results that appeared in the meantime. Most algorithms in the area optimize for modularity. Modularity has recently been proposed [179]. The core idea for modularity is to take coverage, i.e. the fraction of edges covered by clusters, minus the expected value of the same quantity in a network with the same community divisions, but random connections between the vertices. The commonly used formula is as follows: $\text{mod}(\mathcal{C}) := \frac{m(\mathcal{C})}{m} - \frac{1}{4m^2} \sum_{C \in \mathcal{C}} \left(\sum_{v \in C} \deg(v) \right)^2$.

Miller and Eliassi-Rad [167] adapted a dynamic extension of Latent Dirichlet Allocation for dynamic graph clustering. Latent Dirichlet Allocation has been originally proposed for modeling text documents, i.e. the algorithm assumes that a given set of documents can be classified into k topics. This approach has been transferred to graphs [113] and was adapted by the authors for dynamic networks. Aynaud and Guillaume [23] tracked communities between successive snapshots of the input network. They first noted that using standard community detection algorithms results in stability issues, i.e. little modifications of the network can result in wildly different clusterings. Hence, the authors propose a modification of the Louvain method to obtain stable clusterings. This is done by modifying the initialization routine of the Louvain method. By default, the Louvain method starts with each node being in its own clustering. In the modified version of Aynaud and Guillaume, the algorithm keeps the clustering of the previous time step and uses this as a starting point for the Louvain method which results in much more stable clusterings. Bansal et al. [24] also reused the communities from previous time steps. However, their approach is based on greedy agglomeration where two communities are merged at each step to optimize the modularity objective function. The authors improved the efficiency of dynamic graph clustering algorithms by limiting recomputation to regions of the network and merging processes that have been affected by insertion and deletion operations. Görke et al. [96] showed that the structure of minimum s - t -cuts in a graph allows for efficient updates of

clusterings. The algorithm builds on partially updating a specific part of a minimum-cut tree and is able to maintain a clustering fulfilling a provable quality guarantee, i.e. the clusterings computed by the algorithm are guaranteed to yield a certain expansion. To the best of our knowledge, this is the only dynamic graph clustering algorithm that provides such a guarantee. Later, Görke et al. [99, 98] formally introduced the concept of smoothness to compare consecutive clusterings and provided a portfolio of different update strategies for different types of local and global algorithms. Moreover, their fastest algorithm is guaranteed to run in time $\Theta(\log n)$ per update. Their experimental evaluation indicates that dynamically maintaining a clustering of a dynamic random network saves time and at the same time also yields higher modularity than recomputation from scratch. Alvari et al. [13] proposed a dynamic game theory method to tackle the community detection problem in dynamic social networks. Roughly speaking, the authors model the process of community detection as an iterative game performed in a dynamic multiagent environment where each node is an agent who wants to maximize its total utility. In each iteration, an agent can decide to join, switch, leave, or stay in a community. The new utility is then computed by the best outcome of these operations. The authors use neighborhood similarity to measure structural similarity and optimize for modularity. The experimental evaluation is limited to two graphs. Zakrzweska and Bader [248] presented two algorithms that update communities. Their first algorithm is similar to the dynamic greedy agglomeration algorithm by Görke et al. [98]. The second algorithm is a modification of the first one that runs faster. This first is achieved by more stringent backtracking of merges than Görke et al. [98], i.e. merges are only undone if the merge has significantly changed the modularity score of the clustering. Moreover, the authors used a faster agglomeration scheme during update operations that uses information about previous merges to speed up contractions. Recently, Zhuang et al. [254] proposed the DynaMo algorithm which also is a dynamic algorithm for modularity maximization, however the algorithm processes network changes in batches.

3.12 Centralities

We will describe three popular measures to find central nodes in networks in the fully dynamic setting: Katz centrality, betweenness centrality and closeness centrality. The only two theoretical fully dynamic results that we are aware of are due to Pontecorvi and Ramachandran [191], who achieve amortized $\mathcal{O}(\nu^* \cdot \log^2 n)$ update time for betweenness centrality where ν^* bounds the number of distinct edges that lie on shortest paths through any single vertex, and a result due to van den Brand and Nanongkai [233], who present a $(1 + \epsilon)$ -approximate fully-dynamic algorithm for closeness centrality with $\mathcal{O}(n^{1.823})$ update time. This is an obvious area for future work.

Katz Centrality

Katz centrality is a centrality metric that measure the relation between vertices by counting weighted walks between them. Nathan and Bader [177] were the first to look at the problem in a dynamic setting. At that time, static algorithms mostly used linear algebra-based techniques to compute Katz scores. The dynamic version of their algorithm incrementally updates the scores by exploiting properties of iterative solvers, i.e. Jacobi iterations. Their algorithm achieved speedups of over two orders of magnitude over the simple algorithms that perform static recomputation every time the graph changes. Later, they improved their algorithm [176] to handle updates by using an alternative, agglomerative method of calculating Katz scores. While their static algorithm is already several orders of magnitude

faster than typical linear algebra approaches, their dynamic algorithm is also faster than pure static recomputation every time the graph changes. A drawback of the algorithms by Nathan and Bader is that they are unable to reproduce the exact Katz ranking after dynamic updates. Van der Grinten et al. [235] fixed this problem by presenting a dynamic algorithm that iteratively improves upper and lower bounds on the centrality scores. The computed scores are approximate, but the bounds guarantee the correct ranking. The dynamic algorithm improves over the static recomputation of the Katz rankings as long as the size of the batches in the update sequence is smaller than 10 000.

Betweenness Centrality

Given a graph and a vertex v in the graph, the betweenness centrality measure is defined to be $c(v) = \sum_{u,w,u \neq w} \frac{\sigma_{u,w}(v)}{\sigma_{u,w}}$, where $\sigma_{u,w}$ is the number of shortest paths between u and w and $\sigma_{u,w}(v)$ is the number of shortest paths between u and w that include v . Statically computing betweenness centrality involves solving the all-pairs shortest path problem. Dynamically maintaining betweenness centrality is challenging as the insertion or deletion of a single edge can lead to changes of many shortest paths in the graph. The QUBE algorithm [156] was the first to provide a non-trivial update routine. The key idea is to perform the betweenness computation on a reduced set of vertices, i.e. the algorithm first finds vertices whose centrality index might have changed. Betweenness centrality is then only computed on the first set of vertices. However, QUBE is limited to the insertion and deletion of non-bridge edges. Lee et al. [155] extended the QUBE algorithm [156] to be able to insert and delete non-bridge edges. Moreover, the authors reduced the number of shortest paths that need to be recomputed and thus gained additional speedups over QUBE. Kourtellis et al. [152, 151] contributed an algorithm that maintains both vertex and edge betweenness centrality. Their algorithm needs less space than the algorithm by Green et al. [102] as it avoids storing predecessor lists. Their method can be parallelized and runs on top of parallel data processing engines such as Hadoop. Bergamini et al. [35] presented an incremental *approximation* algorithm for the problem which is based on the first theory result that is asymptotically faster than recomputing everything from scratch due to Nasre et al. [175]. As a building block of their algorithm, the authors used an asymptotically faster algorithm for the dynamic single-source shortest path problem and additionally sample shortest paths. Experiments indicate that the algorithm can be up to four orders of magnitude faster compared to restarting the static approximation algorithm by Riondato and Kornaropoulos [197]. In the same year, the authors extended their algorithm to become a fully dynamic approximation algorithm for the problem [33, 34]. In addition to dynamic single-source shortest paths, the authors also employed an approximation of the vertex diameter that is needed to compute the number of shortest paths that need to be sampled as a function of a given error guarantee that should be achieved. Hayashi et al. [112] provided a fully dynamic approximation algorithm that is also based on sampling. In contrast to Bergamini et al. [35, 33, 34], which selects between each pair of sampled vertices, the authors save all the paths between each sampled pair of vertices. Moreover, the shortest paths are represented in a data structure called hypergraph sketch. To further reduce the running time when handling unreachable pairs, the authors maintain a reachability index. Gil-Pons [190] focused on exact betweenness in incremental graphs. The author presented a space-efficient algorithm with linear space complexity. Lastly, Chehreghani et al. [61] focused on the special case in which the betweenness of a single node has to be maintained under updates.

Closeness Centrality

Given a graph and a vertex v , the harmonic closeness centrality measure is defined as $\text{clo}(v) = \sum_{u \in V, u \neq v} \frac{1}{d(u,v)}$ where $d(u,v)$ is the distance between u and v . Roughly speaking, it is the sum of the reciprocal length of the shortest path between the node and all other nodes in the graph. Baevla's definition of closeness centrality is similarly $\frac{|V|-1}{\sum_{v \in V} d(u,v)}$. Kas et al. [144] were the first to give an *fully dynamic* algorithm for the problem. As computing closeness centrality depends on the all-pairs shortest path problem, the authors extended an existing dynamic all-pairs shortest path algorithm [193] for their problem. As the algorithm stores pairwise distances between nodes it has quadratic memory requirement. Sariyuce et al. [209] provided an algorithm that can handle insertions and deletions. In contrast to Kas et al. [144], the authors used static single-source shortest paths from each vertex. The algorithm does not need to store pairwise distances and hence requires only a linear amount of memory. Moreover, the authors observed that in scale-free networks the diameter grows proportional to the logarithm of the number of nodes, i.e. the diameter is typically small. When the graph is modified with minor updates, the diameter also tends to stay small. This can be used to limit the number of vertices that need to be updated. In particular, the authors showed that recomputation of closeness can be skipped for vertices s such that $|d(s,u) - d(s,v)| = 1$ where u, v are the endpoints of the newly inserted edge. Lastly, the authors used data reduction rules to filter vertices, i.e. real-life networks can contain nodes that have the same or similar neighborhood structure that can be merged. Later, Sariyuce et al. [210, 211] proposed a distributed memory-parallel algorithm for the problem. Yen et al. [247] proposed the fully dynamic algorithm CENDY which can reduce the number of internal updates to a few single-source shortest path computations necessary by using breadth-first searches. The main idea is that given an augmented rooted BFS tree of an unweighted network, edges that are inserted or deleted within the same level of the tree do not change the distances from the root to all other vertices. Putman et al. [192] provided a faster algorithm for fully dynamic harmonic closeness. The authors also used a filtering method to heavily reduce the number of computations for each incremental update. The filtering method is an extension of level-based filtering to directed and weighted networks. The dynamic algorithm by Shao et al. [219] maintains closeness centrality by efficiently detecting all affected shortest paths based on articulation points. The main observation is that a graph can be divided into a series of biconnected components which are connected by articulation points – the distances between two arbitrary vertices in the graph can be expressed as multiple distances between different biconnected components.

Bisenius et al. [49] contributed an algorithm to maintain top- k harmonic closeness in fully dynamic graphs. The algorithm is not required to compute closeness centrality for the initial graph and the memory footprint of their algorithm is linear. Their algorithm also tries to skip recomputations of vertices that are unaffected by the modifications of the graph by running breadth-first searches. Crescenzi et al. [63] gave a fully dynamic *approximation* algorithm for top- k harmonic closeness. The algorithm is based on sampling paths and a backward dynamic breadth-first search algorithm.

3.13 Graph Partitioning

Typically the graph partitioning problem asks for a partition of a graph into k blocks of about equal size such that there are few edges between them. More formally, given a graph $G = (V, E)$, we are looking for disjoint *blocks* of vertices V_1, \dots, V_k that partition V , i.e., $V_1 \cup \dots \cup V_k = V$. A *balancing constraint* demands that all blocks have weight

$c(V_i) \leq (1 + \epsilon) \lceil \frac{c(V)}{k} \rceil$ for some imbalance parameter ϵ . The most used objective is to minimize the total *cut* $\omega(E \cap \bigcup_{i < j} V_i \times V_j)$. The problem is known to be NP-hard and no constant-factor approximation algorithms exist. Thus heuristic algorithms are mostly used in practice. Dynamic graph partitioning algorithms are also known in the community as *repartitioning algorithms*. As the problem is typically not solved to optimality in practice, repartitioning involves a tradeoff between the *quality*, i.e. the number of edges in different sets of the partitioning, and the amount of vertices that need to change their block as they cause communication when physically moved between the processors as the partition is adopted. The latter is especially important when graph partitioning is used in adaptive numerical simulations. In these simulations, the main goal is to partition a model of computation and communication in which nodes model computation and edges model communication. The blocks of the partition are then fixed to a specific processing element. When the dynamic graph partitioning algorithm decides to change the blocks due to changes in the graph topology, nodes that are moved to a different block create communication in the simulation system as the underlying data needs to be moved between the corresponding processors.

Hendrikson et al. [114] tackled the repartitioning problem by introducing k virtual vertices. Each of the virtual vertices is connected to all nodes of its corresponding block. The edges get a weight α which is proportional to the migration cost of a vertex and the vertex weights of the virtual vertices are set to zero. Then an updated partition can be computed using a static partitioning algorithm since the model accounts for migration costs and edge cut size at the same time.

Schloegel et al. [213] presented heuristics to control the tradeoff between edge-cut size and vertex migration costs. The most simple algorithm is to compute a completely new partition and then determine a mapping between the blocks of the old and the new partition that minimizes migration. The more sophisticated algorithm of [213] is a multilevel algorithm based on a simple process, i.e. nodes are moved from blocks that contain too many vertices to blocks that contain not enough vertices. However, this approach often yields partitions that cut a large number of edges. The result has been improved later by combining the two approaches in the parallel partitioning tool ParMetis [214]. Schloegel et al. [215] later extended their algorithm to be able to handle multiple balance constraints. Hu and Blake [129] noted that diffusion processes can suffer from slow convergence and improved the performance of diffusion through the use of Chebyshev polynomials. More precisely, the diffusion process in their paper is a directed diffusion that computes a diffusion solution by solving a so-called head conduction equation while minimizing the data movement. Walshaw et al. [240] integrated a repartitioning algorithm into their parallel (meanwhile uncontinued) tool Jostle. The algorithm is a directed diffusion process based on the solver proposed by Hu and Blake [129]. Rotaru and Nägeli [203] extended previous diffusion-based algorithms to be able to handle heterogeneous systems. These approaches, however, have certain weaknesses: For example, in numerical applications the maximum number of boundary nodes of a block is often a better estimate of the occurring communication in the simulation than the number of edges cut. Meyerhenke and Gehweiler [165, 166] explored a disturbed diffusion process that is able to overcome some of the issues of the previous approaches. To do so, Meyerhenke adapted DIBAP, a previously developed algorithm that aims at computing well-shaped partitions. A diffusion process is called disturbed if its convergence state does not result in a balanced distribution. These processes can be helpful to find densely connected regions in the graph.

There has been also work that tackles slightly different problem formulations. Kiefer et al. [146] noted that performance in applications usually does not scale linearly with the amount of work per block due to contention on different compute components. Their

algorithm uses a simplified penalized resource consumption model. Roughly speaking, the authors introduced a penalized block weight and modified the graph partitioning problem accordingly. More precisely, a positive, monotonically increasing penalty function p is used to penalize the weight of a block based on the partition cardinality. Vaquero et al. [237] looked at the problem for distributed graph processing systems. Their approach is based on iterative vertex migration based on label propagation. More precisely, a vertex has a list of candidate blocks where the highest number of its neighbors are located. However, initial partitions are computed using hashing which does not yield high quality partitions since it completely ignores the structure of the graph. The authors did not compare their work against other state-of-the-art repartitioning algorithms, so it is unclear how well the algorithm performs compared to other algorithms. Xu et al. [245] and Nicoara et al. [180] also presented dynamic algorithms specifically designed for graph processing systems. Other approaches have focused on the edge partitioning problem [204, 131, 83] or the special case of road networks [52].

4 Dynamic Graph Systems

The methodology of the previous two sections is to engineer algorithms for specific dynamic graph problems. In contrast to this, there are also approaches that try to engineer dynamic graph systems that can be applied to a wide range of dynamic graph problems. Alberts et al. [11] started this effort and presented a software library of dynamic graph algorithms. The library is written in C++ and is an extension of the well known LEDA library of efficient data types and algorithms. The library contains algorithms for connectivity, spanning trees, single-source shortest paths and transitive closure.

A decade later Weigert et al. [243] presented a system that is able to deal with dynamic distributed graphs, i.e. in settings in which a graph is too large for the memory of a single machine and, thus, needs to be distributed over multiple machines. A user can implement a query function to implement graph queries. Based on their experiments, the system appears to scale well to large distributed graphs. Ediger et al. [77] engineered *STINGER* which is short for Spatio-Temporal Interaction Networks and Graphs Extensible Representation. *STINGER* provides fast insertions, deletions, and updates on semantic graphs that have a skewed degree distribution. The authors showed in their experiments that the system can handle 3 million updates per second on a scale-free graph with 537 million edges on a Cray XMT machine. The authors already implemented a variety of algorithms on *STINGER* including community detection, k -core extraction, and many more. Later, Feng et al. [84] presented *DISTINGER* which has the same goals as *STINGER*, but focuses on the distributed memory case, i.e. the authors presented a distributed graph representation. Vaquero et al. [236] presented a dynamic graph processing system that uses adaptive partitioning to update the graph distribution over the processors over time. This speeds up queries as a better graph distribution significantly reduces communication overhead. Experiments showed that the repartitioning heuristic (also explained in Section 3.13) improves computation performance in their system up to 50% for an algorithm that computes the estimated diameter in a graph. Sengupta et al. [217] introduced a dynamic graph analytics framework called *GraphIn*. Part of *GraphIn* is a new programming model based on the gather-apply-scatter programming paradigm that allows users to implement a wide range of graph algorithms that run in parallel. Compared to *STINGER*, the authors reported a 6.6-fold speedup. Iwabuchi et al. [136] presented an even larger speedup over *STINGER*. Their dynamic graph data store is, like *STINGER*, designed for scale-free networks. The system uses compact hash tables with high data locality. In their experiments, their system called *DegAwareRHH*, is a factor 206.5 faster than *STINGER*.

Another line of research focuses on graph analytic frameworks and data structures for GPUs. Green and Bader [84] presented **cuSTINGER**, which is a GPU extension of **STINGER** and targets NVIDIA’s CUDA supported GPUs. One drawback of **cuSTINGER** is that the system has to perform restarts after a large number of edge updates. Busato et al. fixed this issue in their system, called **Hornet**, and, thus, outperform **cuSTINGER**. Moreover, **Hornet** uses a factor of 5 to 10 less memory than **cuSTINGER**. In contrast to previous approaches, **faimGraph** due to Winter et al. [244] is able to deal with a changing number of vertices. Awad et al. [22] noted that the experiments performed by Busato et al. are missing true dynamism that is expected in real world scenarios and proposed a dynamic graph structure that uses one hash table per vertex to store adjacency lists. The system achieves speedups between 5.8 to 14.8 compared to **Hornet** and 3.4 to 5.4 compared to **faimGraph** for batched edge insertions (and slightly smaller speedups for batched edge deletions). The algorithm also supports vertex deletions, as does **faimGraph**.

5 Methodology

Currently there is a limited amount of real-world *fully* dynamic networks publicly available. There are repositories that feature a lot of real-world insertions only instances such as **SNAP**³ and **KONECT**⁴. However, since the fully dynamic instances are rarely available at the moment, we start a new graph repository that provides fully dynamic graph instances⁵. Currently, there is also very limited work on dynamic graph generators. A generator for clustered dynamic random networks has been proposed by Görke et al. [97]. Another approach is due to Sengupta [218] to generate networks for dynamic overlapping communities in networks. A generative model for dynamic networks with community structure can be found in [30]. This is a widely open topic for future work, both in terms of oblivious adversaries as well as adaptive adversaries. To still be able to evaluate fully dynamic algorithms in practice, research uses a wide range of models at the moment to turn static networks into dynamic ones. We give a brief overview over the most important ones. In *undo-based* approaches, edges of a static network are inserted in some order until all edges are inserted. In the end, $x\%$ of the last insertions are undone. The intuition here is that one wants undo changes that happened to a network and to recreate a previous state of the data structure. In *window-based* approaches, edges are inserted and have a predefined lifetime. That means an edge is deleted after a given number d of new edges have been inserted. In *remove and add* based approaches, a small fraction of random edges from a static network is removed and later on reinserted. In practice, researchers use a single edge as well as whole batches of edges. In *morphing-based* approaches, one takes two related networks and creates a sequence of edge updates such that the second network obtained after the update sequence has been applied to the first network.

³ <https://snap.stanford.edu/>

⁴ <http://konect.cc/>

⁵ <https://DynGraphLab.github.io>

References

- 1 Amir Abboud, Raghavendra Addanki, Fabrizio Grandoni, Debmalya Panigrahi, and Barna Saha. Dynamic set cover: improved algorithms and lower bounds. In Moses Charikar and Edith Cohen, editors, *Proc. of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 114–125. ACM, 2019. doi:10.1145/3313276.3316376.
- 2 Amir Abboud and Søren Dahlgaard. Popular conjectures as a barrier for dynamic planar graph algorithms. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 477–486. IEEE, 2016.
- 3 Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *Foundations of Computer Science (FOCS), 2014 IEEE 55th Annual Symposium on*, pages 434–443. IEEE, 2014. doi:10.1109/FOCS.2014.53.
- 4 Ittai Abraham, Shiri Chechik, and Sebastian Krininger. Fully dynamic all-pairs shortest paths with worst-case update-time revisited. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 440–452. SIAM, 2017. doi:10.1137/1.9781611974782.28.
- 5 Ittai Abraham, Shiri Chechik, and Kunal Talwar. Fully dynamic all-pairs shortest paths: Breaking the $o(n)$ barrier. In Klaus Jansen, José D. P. Rolim, Nikhil R. Devanur, and Cristopher Moore, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2014, September 4-6, 2014, Barcelona, Spain*, volume 28 of *LIPICs*, pages 1–16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014. doi:10.4230/LIPICs.APPROX-RANDOM.2014.1.
- 6 Umut A. Acar, Daniel Anderson, Guy E. Blelloch, and Laxman Dhulipala. Parallel batch-dynamic graph connectivity. In Christian Scheideler and Petra Berenbrink, editors, *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2019, Phoenix, AZ, USA, June 22-24, 2019*, pages 381–392. ACM, 2019. doi:10.1145/3323165.3323196.
- 7 Umut A. Acar, Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, and Sam Westrick. Parallel batch-dynamic trees via change propagation. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 173 of *LIPICs*, pages 2:1–2:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.ESA.2020.2.
- 8 Nesreen K. Ahmed, Nick G. Duffield, Theodore L. Willke, and Ryan A. Rossi. On sampling from massive graph streams. *Proc. VLDB Endow.*, 10(11):1430–1441, 2017. doi:10.14778/3137628.3137651.
- 9 Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In Chin-Wan Chung, Andrei Z. Broder, Kyuseok Shim, and Torsten Suel, editors, *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014*, pages 237–248. ACM, 2014. doi:10.1145/2566486.2568007.
- 10 David Alberts, Giuseppe Cattaneo, and Giuseppe F. Italiano. An empirical study of dynamic graph algorithms. *ACM J. Exp. Algorithmics*, 2:5, 1997. doi:10.1145/264216.264223.
- 11 David Alberts, Giuseppe Cattaneo, Giuseppe F Italiano, Umberto Nanni, and Christos Zaroliagis. A software library of dynamic graph algorithms. In *Proc. Workshop on Algorithms and Experiments*, pages 129–136. Citeseer, 1998.
- 12 Bowen Alpern, Roger Hoover, Barry K. Rosen, Peter F. Sweeney, and F. Kenneth Zadeck. Incremental evaluation of computational circuits. In David S. Johnson, editor, *Proc. of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California, USA*, pages 32–42. SIAM, 1990. URL: <http://dl.acm.org/citation.cfm?id=320176.320180>.

- 13 Hamidreza Alvari, Alireza Hajibagheri, and Gita Reese Sukthankar. Community detection in dynamic social networks: A game-theoretic approach. In Xindong Wu, Martin Ester, and Guandong Xu, editors, *2014 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining, ASONAM 2014, Beijing, China, August 17-20, 2014*, pages 101–107. IEEE Computer Society, 2014. doi:10.1109/ASONAM.2014.6921567.
- 14 Abhash Anand, Surender Baswana, Manoj Gupta, and Sandeep Sen. Maintaining approximate maximum weighted matching in fully dynamic graphs. In Deepak D’Souza, Telikepalli Kavitha, and Jaikumar Radhakrishnan, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2012, December 15-17, 2012, Hyderabad, India*, volume 18 of *LIPICs*, pages 257–266. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPICs.FSTTCS.2012.257.
- 15 Bertie Ancona, Monika Henzinger, Liam Roditty, Virginia Vassilevska Williams, and Nicole Wein. Algorithms and hardness for diameter in dynamic graphs. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPICs*, pages 13:1–13:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.ICALP.2019.13.
- 16 Eugenio Angriman, Henning Meyerhenke, Christian Schulz, and Bora Uçar. Fully-dynamic weighted matching approximation in practice. *CoRR*, abs/2104.13098, 2021. arXiv:2104.13098.
- 17 Moab Arar, Shiri Chechik, Sarel Cohen, Cliff Stein, and David Wajc. Dynamic matching: Reducing integral algorithms to approximately-maximal fractional algorithms. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018*, pages 7:1–7:16, 2018. doi:10.4230/LIPICs.ICALP.2018.7.
- 18 Sabeur Aridhi, Martin Brugnara, Alberto Montresor, and Yannis Velegrakis. Distributed k-core decomposition and maintenance in large dynamic graphs. In *Proc. of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 161–168, 2016. doi:10.1145/2933267.2933299.
- 19 Sepehr Assadi, Krzysztof Onak, Baruch Schieber, and Shay Solomon. Fully dynamic maximal independent set with sublinear update time. In Ilias Diakonikolas, David Kempe, and Monika Henzinger, editors, *Proc. of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 815–826. ACM, 2018. doi:10.1145/3188745.3188922.
- 20 Sepehr Assadi, Krzysztof Onak, Baruch Schieber, and Shay Solomon. Fully dynamic maximal independent set with sublinear in n update time. In Timothy M. Chan, editor, *Proc. of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1919–1936. SIAM, 2019. doi:10.1137/1.9781611975482.116.
- 21 G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and their Approximability Properties*. Springer Science & Business Media, 2012.
- 22 Muhammad A. Awad, Saman Ashkiani, Serban D. Porumbescu, and John D. Owens. Dynamic graphs on the GPU. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, May 18-22, 2020*, pages 739–748. IEEE, 2020. doi:10.1109/IPDPS47924.2020.00081.
- 23 Thomas Aynaud and Jean-Loup Guillaume. Static community detection algorithms for evolving networks. In Lavy Libman Ariel Orda, Nidhi Hegde, editor, *8th International Symposium on Modeling and Optimization in Mobile, Ad-Hoc and Wireless Networks (WiOpt 2010), May 31 - June 4, 2010, University of Avignon, Avignon, France*, pages 513–519. IEEE, 2010. URL: <http://ieeexplore.ieee.org/document/5520221/>.

- 24 Shweta Bansal, Sanjukta Bhowmick, and Prashant Paymal. Fast community detection for dynamic complex networks. In Luciano da F. Costa, Alexandre G. Evsukoff, Giuseppe Mangioni, and Ronaldo Menezes, editors, *Complex Networks - Second International Workshop, CompleNet 2010, Rio de Janeiro, Brazil, October 13-15, 2010, Revised Selected Papers*, volume 116 of *Communications in Computer and Information Science*, pages 196–207. Springer, 2010. doi:10.1007/978-3-642-25501-4_20.
- 25 Surender Baswana, Shreejit Ray Chaudhury, Keerti Choudhary, and Shahbaz Khan. Dynamic DFS in undirected graphs: Breaking the $o(m)$ barrier. *SIAM J. Comput.*, 48(4):1335–1363, 2019. doi:10.1137/17M114306X.
- 26 Surender Baswana, Manoj Gupta, and Sandeep Sen. Fully dynamic maximal matching in $O(\log n)$ update time. *SIAM J. Comput.*, 44(1):88–113, 2015. doi:10.1137/16M1106158.
- 27 Surender Baswana, Shiv Kumar Gupta, and Ayush Tulsyan. Fault tolerant and fully dynamic DFS in undirected graphs: Simple yet efficient. In Peter Rossmanith, Pinar Heggernes, and Joost-Pieter Katoen, editors, *44th International Symposium on Mathematical Foundations of Computer Science, MFCS 2019, August 26-30, 2019, Aachen, Germany*, volume 138 of *LIPICs*, pages 65:1–65:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.MFCS.2019.65.
- 28 Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining hierarchical and goal-directed speed-up techniques for dijkstra’s algorithm. *ACM J. Exp. Algorithmics*, 15, 2010. doi:10.1145/1671970.1671976.
- 29 Reinhard Bauer and Dorothea Wagner. Batch dynamic single-source shortest-path algorithms: An experimental study. In Jan Vahrenhold, editor, *Experimental Algorithms, 8th International Symposium, SEA 2009, Dortmund, Germany, June 4-6, 2009. Proc.*, volume 5526 of *Lecture Notes in Computer Science*, pages 51–62. Springer, 2009. doi:10.1007/978-3-642-02011-7_7.
- 30 F. Becker. Generative Model for Dynamic Networks with Community Structures. Master’s Thesis, Heidelberg University, 2020.
- 31 Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Cliff Stein, and Madhu Sudan. Fully dynamic maximal independent set with polylogarithmic update time. In David Zuckerman, editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 382–405. IEEE Computer Society, 2019. doi:10.1109/FOCS.2019.00032.
- 32 Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.
- 33 Elisabetta Bergamini and Henning Meyerhenke. Fully-dynamic approximation of betweenness centrality. In Nikhil Bansal and Irene Finocchi, editors, *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proc.*, volume 9294 of *Lecture Notes in Computer Science*, pages 155–166. Springer, 2015. doi:10.1007/978-3-662-48350-3_14.
- 34 Elisabetta Bergamini and Henning Meyerhenke. Approximating betweenness centrality in fully dynamic networks. *Internet Math.*, 12(5):281–314, 2016. doi:10.1080/15427951.2016.1177802.
- 35 Elisabetta Bergamini, Henning Meyerhenke, and Christian Staudt. Approximating betweenness centrality in large evolving networks. In Ulrik Brandes and David Eppstein, editors, *Proc. of the Seventeenth Workshop on Algorithm Engineering and Experiments, ALENEX 2015, San Diego, CA, USA, January 5, 2015*, pages 133–146. SIAM, 2015. doi:10.1137/1.9781611973754.12.
- 36 Aaron Bernstein, Sebastian Forster, and Monika Henzinger. A deamortization approach for dynamic spanner and dynamic maximal matching. In Timothy M. Chan, editor, *Proc. of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1899–1918. SIAM, 2019. doi:10.1137/1.9781611975482.115.
- 37 Aaron Bernstein and Cliff Stein. Faster fully dynamic matchings with small approximation ratios. In *Proc. of the 27th Symposium on Discrete Algorithms SODA*, pages 692–711. SIAM, 2016. doi:10.1137/1.9781611974331.ch50.

- 38 Emanuele Berrettini, Gianlorenzo D’Angelo, and Daniel Delling. Arc-flags in dynamic graphs. In Jens Clausen and Gabriele Di Stefano, editors, *ATMOS 2009 - 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems*, IT University of Copenhagen, Denmark, September 10, 2009, volume 12 of *OASICS*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009. URL: <http://drops.dagstuhl.de/opus/volltexte/2009/2149>.
- 39 Sayan Bhattacharya, Deeparnab Chakrabarty, and Monika Henzinger. Deterministic dynamic matching in $O(1)$ update time. *Algorithmica*, 82(4):1057–1080, 2020. doi:10.1007/s00453-019-00630-4.
- 40 Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Deterministic fully dynamic data structures for vertex cover and matching. *SIAM J. Comput.*, 47(3):859–887, 2018. doi:10.1137/140998925.
- 41 Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Dynamic algorithms via the primal-dual method. *Inf. Comput.*, 261:219–239, 2018. doi:10.1016/j.ic.2018.02.005.
- 42 Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. New deterministic approximation algorithms for fully dynamic matching. In *Proc. of the 48th Annual Symposium on Theory of Computing*, pages 398–411. ACM, 2016. doi:10.1145/2897518.2897568.
- 43 Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. Fully dynamic approximate maximum matching and minimum vertex cover in $O(\log^3 n)$ worst case update time. In Philip N. Klein, editor, *Proc. of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms SODA*, pages 470–489. SIAM, 2017. doi:10.1137/1.9781611973730.54.
- 44 Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. A new deterministic algorithm for dynamic set cover. In David Zuckerman, editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 406–423. IEEE Computer Society, 2019. doi:10.1109/FOCS.2019.00033.
- 45 Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos E. Tsourakakis. Space- and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *Proc. of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 173–182. ACM, 2015. doi:10.1145/2746539.2746592.
- 46 Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Xiaowei Wu. An improved algorithm for dynamic set cover. *CoRR*, abs/2002.11171, 2020. arXiv:2002.11171.
- 47 Sayan Bhattacharya and Janardhan Kulkarni. Deterministically maintaining a $(2 + \epsilon)$ -approximate minimum vertex cover in $o(1/\epsilon^2)$ amortized update time. In Timothy M. Chan, editor, *Proc. of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1872–1885. SIAM, 2019. doi:10.1137/1.9781611975482.113.
- 48 Sujoy Bhowmik, Guangping Li, and Martin Nöllenburg. An algorithmic study of fully dynamic independent sets for map labeling. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 173 of *LIPICs*, pages 19:1–19:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.ESA.2020.19.
- 49 Patrick Bisenius, Elisabetta Bergamini, Eugenio Angriman, and Henning Meyerhenke. Computing top- k closeness centrality in fully-dynamic graphs. In Rasmus Pagh and Suresh Venkatasubramanian, editors, *Proc. of the Twentieth Workshop on Algorithm Engineering and Experiments, ALENEX 2018, New Orleans, LA, USA, January 7-8, 2018*, pages 21–35. SIAM, 2018. doi:10.1137/1.9781611975055.3.
- 50 Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(9):1124–1137, 2004. doi:10.1109/TPAMI.2004.60.

- 51 U. Brandes, D. Delling, M. Gaertler, R. Gorke, M. Hoefer, Z. Nikoloski, and D. Wagner. On Modularity Clustering. *IEEE Transactions on Knowledge and Data Engineering*, 20(2):172–188, 2008. doi:10.1109/TKDE.2007.190689.
- 52 Valentin Buchhold, Daniel Delling, Dennis Schieferdecker, and Michael Wegner. Fast and stable repartitioning of road networks. In *18th International Symposium on Experimental Algorithms (SEA 2020)*, volume 160 of *LIPICs*, pages 26:1–26:15. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.SEA.2020.26.
- 53 Laurent Bulteau, Vincent Froese, Konstantin Kutzkov, and Rasmus Pagh. Triangle counting in dynamic graph streams. *Algorithmica*, 76(1):259–278, 2016. doi:10.1007/s00453-015-0036-4.
- 54 Luciana S. Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. Counting triangles in data streams. In Stijn Vansummeren, editor, *Proc. of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 26-28, 2006, Chicago, Illinois, USA*, pages 253–262. ACM, 2006. doi:10.1145/1142351.1142388.
- 55 Luciana S. Buriol, Mauricio G. C. Resende, and Mikkel Thorup. Speeding up dynamic shortest-path algorithms. *INFORMS J. Comput.*, 20(2):191–204, 2008. doi:10.1287/ijoc.1070.0231.
- 56 Giuseppe Cattaneo, Pompeo Faruolo, Umberto Ferraro Petrillo, and Giuseppe F. Italiano. Maintaining dynamic minimum spanning trees: An experimental study. In David M. Mount and Clifford Stein, editors, *Algorithm Engineering and Experiments, 4th International Workshop, ALENEX 2002, San Francisco, CA, USA, January 4-5, 2002, Revised Papers*, volume 2409 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2002. doi:10.1007/3-540-45643-0_9.
- 57 Giuseppe Cattaneo, Pompeo Faruolo, Umberto Ferraro Petrillo, and Giuseppe F. Italiano. Maintaining dynamic minimum spanning trees: An experimental study. *Discret. Appl. Math.*, 158(5):404–425, 2010. doi:10.1016/j.dam.2009.10.005.
- 58 Edward P. F. Chan and Yaya Yang. Shortest path tree computation in dynamic graphs. *IEEE Trans. Computers*, 58(4):541–557, 2009. doi:10.1109/TC.2008.198.
- 59 Moses Charikar and Shay Solomon. Fully dynamic almost-maximal matching: Breaking the polynomial worst-case time barrier. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018*, volume 107 of *LIPICs*, pages 33:1–33:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICs.ICALP.2018.33.
- 60 Shiri Chechik and Tianyi Zhang. Fully dynamic maximal independent set in expected poly-log update time. In David Zuckerman, editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 370–381. IEEE Computer Society, 2019. doi:10.1109/FOCS.2019.00031.
- 61 Mostafa Haghiri Chehreghani, Albert Bifet, and Talel Abdesslem. Dybed: An efficient algorithm for updating betweenness centrality in directed dynamic graphs. In Naoki Abe, Huan Liu, Calton Pu, Xiaohua Hu, Nesreen K. Ahmed, Mu Qiao, Yang Song, Donald Kossmann, Bing Liu, Kisung Lee, Jiliang Tang, Jingrui He, and Jeffrey S. Saltz, editors, *IEEE International Conference on Big Data, Big Data 2018, Seattle, WA, USA, December 10-13, 2018*, pages 2114–2123. IEEE, 2018. doi:10.1109/BigData.2018.8622452.
- 62 Li Chen, Gramoz Goranci, Monika Henzinger, Richard Peng, and Thatchaphol Saranurak. Fast dynamic cuts, distances and effective resistances via vertex sparsifiers. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 1135–1146. IEEE, 2020. doi:10.1109/FOCS46700.2020.00109.
- 63 Pierluigi Crescenzi, Clémence Magnien, and Andrea Marino. Finding top-k nodes for temporal closeness in large temporal graphs. *Algorithms*, 13(9):211, 2020. doi:10.3390/a13090211.
- 64 Michael Crouch and Daniel S. Stubbs. Improved streaming algorithms for weighted matching, via unweighted matching. In Klaus Jansen, José D. P. Rolim, Nikhil R. Devanur, and Christopher Moore, editors, *Approximation, Randomization, and Combinatorial Optimization*.

- Algorithms and Techniques, APPROX/RANDOM 2014, September 4-6, 2014, Barcelona, Spain*, volume 28 of *LIPICs*, pages 96–104. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014. doi:10.4230/LIPICs.APPROX-RANDOM.2014.96.
- 65 Annalisa D’Andrea, Mattia D’Emidio, Daniele Frigioni, Stefano Leucci, and Guido Proietti. Dynamic maintenance of a shortest-path tree on homogeneous batches of updates: New algorithms and experiments. *ACM J. Exp. Algorithmics*, 20:1.5:1.1–1.5:1.33, 2015. doi:10.1145/2786022.
- 66 Gianlorenzo D’Angelo, Mattia D’Emidio, and Daniele Frigioni. Fully dynamic update of arc-flags. *Networks*, 63(3):243–259, 2014. doi:10.1002/net.21542.
- 67 Gianlorenzo D’Angelo, Mattia D’Emidio, and Daniele Frigioni. Fully dynamic 2-hop cover labeling. *ACM J. Exp. Algorithmics*, 24(1):1.6:1–1.6:36, 2019. doi:10.1145/3299901.
- 68 Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato Fonseca F. Werneck. Customizable route planning. In Panos M. Pardalos and Steffen Rebennack, editors, *Experimental Algorithms - 10th International Symposium, SEA 2011, Kolimpari, Chania, Crete, Greece, May 5-7, 2011. Proc.*, volume 6630 of *Lecture Notes in Computer Science*, pages 376–387. Springer, 2011. doi:10.1007/978-3-642-20662-7_32.
- 69 Daniel Delling and Dorothea Wagner. Landmark-based routing in dynamic graphs. In Camil Demetrescu, editor, *Experimental Algorithms, 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007, Proc.*, volume 4525 of *Lecture Notes in Computer Science*, pages 52–65. Springer, 2007. doi:10.1007/978-3-540-72845-0_5.
- 70 Camil Demetrescu. *Fully Dynamic Algorithms for Path Problems on Directed Graphs*. PhD thesis, Universtita Degli Studi Di Roma, 2001. URL: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.21.8921>.
- 71 Camil Demetrescu, Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Maintaining shortest paths in digraphs with arbitrary arc weights: An experimental study. In Stefan Näher and Dorothea Wagner, editors, *Algorithm Engineering, 4th International Workshop, WAE 2000, Saarbrücken, Germany, September 5-8, 2000, Proc.*, volume 1982 of *Lecture Notes in Computer Science*, pages 218–229. Springer, 2000. doi:10.1007/3-540-44691-5_19.
- 72 Camil Demetrescu and Giuseppe F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004. doi:10.1145/1039488.1039492.
- 73 Camil Demetrescu and Giuseppe F. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. *ACM Trans. Algorithms*, 2(4):578–601, 2006. doi:10.1145/1198513.1198519.
- 74 Laxman Dhulipala, Quanquan C. Liu, and Julian Shun. Parallel batch-dynamic k-clique counting. *CoRR*, abs/2003.13585, 2020. arXiv:2003.13585.
- 75 Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. doi:10.1007/BF01386390.
- 76 Christof Doll, Tanja Hartmann, and Dorothea Wagner. Fully-dynamic hierarchical graph clustering using cut trees. In *12th Intl. Symp. on Algorithms and Data Structures, WADS’11*, volume 6844 of *LNCS*, pages 338–349, 2011. doi:10.1007/978-3-642-22300-6_29.
- 77 David Ediger, Robert McColl, E. Jason Riedy, and David A. Bader. STINGER: high performance data structure for streaming graphs. In *IEEE Conference on High Performance Extreme Computing, HPEC 2012, Waltham, MA, USA, September 10-12, 2012*, pages 1–5. IEEE, 2012. doi:10.1109/HPEC.2012.6408680.
- 78 Jack R. Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972. doi:10.1145/321694.321699.
- 79 David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Thomas H. Spencer. Separator based sparsification. i. planary testing and minimum spanning trees. *J. Comput. Syst. Sci.*, 52(1):3–27, 1996. doi:10.1006/jcss.1996.0002.
- 80 David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Thomas H. Spencer. Separator-based sparsification II: edge and vertex connectivity. *SIAM J. Comput.*, 28(1):341–381, 1998. doi:10.1137/S0097539794269072.

- 81 David Eppstein, Michael T. Goodrich, Darren Strash, and Lowell Trott. Extended dynamic subgraph statistics using h-index parameterized data structures. *Theor. Comput. Sci.*, 447:44–52, 2012. doi:10.1016/j.tcs.2011.11.034.
- 82 David Eppstein and Emma S. Spiro. The h-index of a graph and its application to dynamic subgraph statistics. *J. Graph Algorithms Appl.*, 16(2):543–567, 2012. doi:10.7155/jgaa.00273.
- 83 Wenfei Fan, Muyang Liu, Chao Tian, Ruiqi Xu, and Jingren Zhou. Incrementalization of graph partitioning algorithms. *Proc. VLDB Endow.*, 13(8):1261–1274, 2020. doi:10.14778/3389133.3389142.
- 84 Guoyao Feng, Xiao Meng, and Khaled Ammar. Distinguer: A distributed graph data structure for massive dynamic graph processing. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 1814–1822. IEEE, 2015. doi:10.1109/BigData.2015.7363954.
- 85 Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.*, 14(4):781–798, 1985. doi:10.1137/0214055.
- 86 Greg N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM J. Comput.*, 26(2):484–538, 1997. doi:10.1137/S0097539792226825.
- 87 Daniele Frigioni, Mario Ioffreda, Umberto Nanni, and Giulio Pasqualone. Experimental analysis of dynamic algorithms for the single-source shortest-path problem. *ACM J. Exp. Algorithmics*, 3:5, 1998. doi:10.1145/297096.297147.
- 88 Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Fully dynamic output bounded single source shortest path problem (extended abstract). In Éva Tardos, editor, *Proc. of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, 28-30 January 1996, Atlanta, Georgia, USA*, pages 212–221. ACM/SIAM, 1996. URL: <http://dl.acm.org/citation.cfm?id=313852.313926>.
- 89 Daniele Frigioni, Tobias Miller, Umberto Nanni, and Christos D. Zaroliagis. An experimental study of dynamic algorithms for transitive closure. *ACM J. Exp. Algorithmics*, 6:9, 2001. doi:10.1145/945394.945403.
- 90 Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A search meets graph theory. In *Proc. of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, pages 156–165. SIAM, 2005. URL: <http://dl.acm.org/citation.cfm?id=1070432.1070455>.
- 91 Andrew V. Goldberg, Sagi Hed, Haim Kaplan, Pushmeet Kohli, Robert Endre Tarjan, and Renato F. Werneck. Faster and more dynamic maximum flow by incremental breadth-first search. In Nikhil Bansal and Irene Finocchi, editors, *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proc.*, volume 9294 of *Lecture Notes in Computer Science*, pages 619–630. Springer, 2015. doi:10.1007/978-3-662-48350-3_52.
- 92 Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988.
- 93 Andrew V. Goldberg and Robert Endre Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988. doi:10.1145/48014.61051.
- 94 Gramoz Goranci, Monika Henzinger, and Mikkel Thorup. Incremental exact min-cut in polylogarithmic amortized update time. *ACM Transactions on Algorithms (TALG)*, 14(2):1–21, 2018.
- 95 Gramoz Goranci, Harald Räcke, Thatchaphol Saranurak, and Zihan Tan. The expander hierarchy and its applications to dynamic graph algorithms. *CoRR*, abs/2005.02369, 2020. arXiv:2005.02369.
- 96 Robert Görke, Tanja Hartmann, and Dorothea Wagner. Dynamic graph clustering using minimum-cut trees. *J. Graph Algorithms Appl.*, 16(2):411–446, 2012. doi:10.7155/jgaa.00269.

- 97 Robert Görke, Roland Kluge, Andrea Schumm, Christian Staudt, and Dorothea Wagner. An efficient generator for clustered dynamic random networks. In Guy Even and Dror Rawitz, editors, *Design and Analysis of Algorithms - First Mediterranean Conference on Algorithms, MedAlg 2012, Kibbutz Ein Gedi, Israel, December 3-5, 2012. Proc.*, volume 7659 of *Lecture Notes in Computer Science*, pages 219–233. Springer, 2012. doi:10.1007/978-3-642-34862-4_16.
- 98 Robert Görke, Pascal Maillard, Andrea Schumm, Christian Staudt, and Dorothea Wagner. Dynamic graph clustering combining modularity and smoothness. *ACM J. Exp. Algorithmics*, 18, 2013. doi:10.1145/2444016.2444021.
- 99 Robert Görke, Pascal Maillard, Christian Staudt, and Dorothea Wagner. Modularity-driven clustering of dynamic graphs. In Paola Festa, editor, *Experimental Algorithms, 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proc.*, volume 6049 of *Lecture Notes in Computer Science*, pages 436–448. Springer, 2010. doi:10.1007/978-3-642-13193-6_37.
- 100 Fabrizio Grandoni, Stefano Leonardi, Piotr Sankowski, Chris Schwiegelshohn, and Shay Solomon. $(1 + \epsilon)$ -approximate incremental matching in constant deterministic amortized time. In Timothy M. Chan, editor, *Proc. of the 20th Symposium on Discrete Algorithms*, pages 1886–1898. SIAM, 2019. doi:10.1137/1.9781611975482.114.
- 101 Sergio Greco, Cristian Molinaro, Chiara Pulice, and Ximena Quintana. Incremental maximum flow computation on evolving networks. In Ahmed Seffah, Birgit Penzenstadler, Carina Alves, and Xin Peng, editors, *Proc. of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*, pages 1061–1067. ACM, 2017. doi:10.1145/3019612.3019816.
- 102 Oded Green, Robert McColl, and David A. Bader. A fast algorithm for streaming betweenness centrality. In *2012 International Conference on Privacy, Security, Risk and Trust, PASSAT 2012, and 2012 International Conference on Social Computing, SocialCom 2012, Amsterdam, Netherlands, September 3-5, 2012*, pages 11–20. IEEE Computer Society, 2012. doi:10.1109/SocialCom-PASSAT.2012.37.
- 103 Manoj Gupta and Shahbaz Khan. Simple dynamic algorithms for maximal independent set and other problems. In *4th Symposium on Simplicity in Algorithms, SOSA@SODA 2021*, to appear, 2021. arXiv:1804.01823.
- 104 Manoj Gupta and Richard Peng. Fully dynamic $(1 + \epsilon)$ -approximate matchings. In *54th Symposium on Foundations of Computer Science, FOCS*, pages 548–557. IEEE Computer Society, 2013. URL: <https://ieeexplore.ieee.org/xpl/conhome/6685222/proceeding>.
- 105 Manoj Gupta and Richard Peng. Fully dynamic $(1 + \epsilon)$ -approximate matchings. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 548–557. IEEE Computer Society, 2013.
- 106 Maximilian Probst Gutenberg and Christian Wulff-Nilsen. Fully-dynamic all-pairs shortest paths: Improved worst-case time and space bounds. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 2562–2574. SIAM, 2020. doi:10.1137/1.9781611975994.156.
- 107 Guyue Han and Harish Sethu. Edge sample and discard: A new algorithm for counting triangles in large dynamic graphs. In Jana Diesner, Elena Ferrari, and Guandong Xu, editors, *Proc. of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017, Sydney, Australia, July 31 - August 03, 2017*, pages 44–49. ACM, 2017. doi:10.1145/3110025.3110061.
- 108 Kathrin Hanauer, Monika Henzinger, and Qi Cheng Hua. Fully dynamic four-vertex subgraph counting. *CoRR*, abs/2106.15524, 2021. arXiv:2106.15524.
- 109 Kathrin Hanauer, Monika Henzinger, and Christian Schulz. Faster fully dynamic transitive closure in practice. In Simone Faro and Domenico Cantone, editors, *18th International Symposium on Experimental Algorithms, SEA 2020, June 16-18, 2020, Catania, Italy*, volume 160 of *LIPICs*, pages 14:1–14:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.SEA.2020.14.

- 110 Kathrin Hanauer, Monika Henzinger, and Christian Schulz. Fully dynamic single-source reachability in practice: An experimental study. In Guy E. Blelloch and Irene Finocchi, editors, *Proc. of the Symposium on Algorithm Engineering and Experiments, ALLENEX 2020, Salt Lake City, UT, USA, January 5-6, 2020*, pages 106–119. SIAM, 2020. doi:10.1137/1.9781611976007.9.
- 111 Takanori Hayashi, Takuya Akiba, and Ken-ichi Kawarabayashi. Fully dynamic shortest-path distance query acceleration on massive networks. In Snehasis Mukhopadhyay, ChengXiang Zhai, Elisa Bertino, Fabio Crestani, Javed Mostafa, Jie Tang, Luo Si, Xiaofang Zhou, Yi Chang, Yunyao Li, and Parikshit Sondhi, editors, *Proc. of the 25th ACM International Conference on Information and Knowledge Management, CIKM 2016, Indianapolis, IN, USA, October 24-28, 2016*, pages 1533–1542. ACM, 2016. doi:10.1145/2983323.2983731.
- 112 Takanori Hayashi, Takuya Akiba, and Yuichi Yoshida. Fully dynamic betweenness centrality maintenance on massive networks. *Proc. VLDB Endow.*, 9(2):48–59, 2015. doi:10.14778/2850578.2850580.
- 113 Keith Henderson and Tina Eliassi-Rad. Applying latent dirichlet allocation to group discovery in large graphs. In Sung Y. Shin and Sascha Ossowski, editors, *Proc. of the 2009 ACM Symposium on Applied Computing (SAC), Honolulu, Hawaii, USA, March 9-12, 2009*, pages 1456–1461. ACM, 2009. doi:10.1145/1529282.1529607.
- 114 Bruce Hendrickson, Robert W. Leland, and Rafael Van Driessche. Enhancing data locality by using terminal propagation. In *29th Annual Hawaii International Conference on System Sciences (HICSS-29), January 3-6, 1996, Maui, Hawaii, USA*, pages 565–574. IEEE Computer Society, 1996. doi:10.1109/HICSS.1996.495507.
- 115 Monika Henzinger. The state of the art in dynamic graph algorithms. In *44th Intl. Conf. on Current Trends in Theory and Practice of Computer Science, SOFSEM'18*, volume 10706 of *LNCS*, pages 40–44. Springer, 2018. doi:10.1007/978-3-319-73117-9_3.
- 116 Monika Henzinger, Shahbaz Khan, Richard Paul, and Christian Schulz. Dynamic matching algorithms in practice. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 173 of *LIPICs*, pages 58:1–58:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.ESA.2020.58.
- 117 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proc. of the forty-seventh annual ACM symposium on Theory of computing*, pages 21–30, 2015. doi:10.1145/2746539.2746609.
- 118 Monika Henzinger, Alexander Noe, and Christian Schulz. Practical fully dynamic minimum cut algorithms. *CoRR*, abs/2101.05033, 2021. arXiv:2101.05033.
- 119 Monika R Henzinger and Valerie King. Maintaining minimum spanning trees in dynamic graphs. In *International Colloquium on Automata, Languages, and Programming*, pages 594–604. Springer, 1997.
- 120 Monika Rauch Henzinger. Approximating minimum cuts under insertions. In *International Colloquium on Automata, Languages, and Programming*, pages 280–291. Springer, 1995.
- 121 Monika Rauch Henzinger. Fully dynamic biconnectivity in graphs. *Algorithmica*, 13(6):503–538, 1995. doi:10.1007/BF01189067.
- 122 Monika Rauch Henzinger. Improved data structures for fully dynamic biconnectivity. *SIAM J. Comput.*, 29(6):1761–1815, 2000. doi:10.1137/S0097539794263907.
- 123 Monika Rauch Henzinger and Michael L. Fredman. Lower bounds for fully dynamic connectivity problems in graphs. *Algorithmica*, 22(3):351–362, 1998. doi:10.1007/PL00009228.
- 124 Monika Rauch Henzinger and Valerie King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In Frank Thomson Leighton and Allan Borodin, editors, *Proc. of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, 29 May-1 June 1995, Las Vegas, Nevada, USA*, pages 519–527. ACM, 1995. doi:10.1145/225058.225269.

- 125 Monika Rauch Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, 1999. doi:10.1145/320211.320215.
- 126 Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In Jeffrey Scott Vitter, editor, *Proc. of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pages 79–89. ACM, 1998. doi:10.1145/276698.276715.
- 127 Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001. doi:10.1145/502090.502095.
- 128 Jacob Holm, Eva Rotenberg, and Christian Wulff-Nilsen. Faster fully-dynamic minimum spanning forest. In Nikhil Bansal and Irene Finocchi, editors, *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proc.*, volume 9294 of *Lecture Notes in Computer Science*, pages 742–753. Springer, 2015. doi:10.1007/978-3-662-48350-3_62.
- 129 Y.F. Hu and R.J. Blake. An improved diffusion algorithm for dynamic load balancing. *Parallel Computing*, 25(4):417–444, 1999. doi:10.1016/S0167-8191(99)00002-2.
- 130 Qiang-Sheng Hua, Yuliang Shi, Dongxiao Yu, Hai Jin, Jiguo Yu, Zhipeng Cai, Xiuzhen Cheng, and Hanhua Chen. Faster parallel core maintenance algorithms in dynamic graphs. *IEEE Trans. Parallel Distributed Syst.*, 31(6):1287–1300, 2020. doi:10.1109/TPDS.2019.2960226.
- 131 Jiewen Huang and Daniel Abadi. LEOPARD: lightweight edge-oriented partitioning and replication for dynamic graphs. *Proc. VLDB Endow.*, 9(7):540–551, 2016. doi:10.14778/2904483.2904486.
- 132 Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie. Fully dynamic connectivity in $O(\log n(\log \log n)^2)$ amortized expected time. In Philip N. Klein, editor, *Proc. of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 510–520. SIAM, 2017. doi:10.1137/1.9781611974782.32.
- 133 Giuseppe Amato II, Giuseppe Cattaneo, and Giuseppe F. Italiano. Experimental analysis of dynamic minimum spanning tree algorithms (extended abstract). In Michael E. Saks, editor, *Proc. of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, 5-7 January 1997, New Orleans, Louisiana, USA*, pages 314–323. ACM/SIAM, 1997. URL: <http://dl.acm.org/citation.cfm?id=314161.314314>.
- 134 Giuseppe F. Italiano. Fully dynamic higher connectivity. In *Encyclopedia of Algorithms*, pages 797–800. Springer, 2016. doi:10.1007/978-1-4939-2864-4_154.
- 135 Zoran Ivkovic and Errol L. Lloyd. Fully dynamic maintenance of vertex cover. In *19th International Workshop Graph-Theoretic Concepts in Computer Science*, volume 790 of *LNCS*, pages 99–111, 1993.
- 136 Keita Iwabuchi, Scott Sallinen, Roger Pearce, Brian Van Essen, Maya Gokhale, and Satoshi Matsuoka. Towards a distributed large-scale dynamic graph data store. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 892–901. IEEE, 2016. doi:10.1109/IPDPSW.2016.189.
- 137 Raj Iyer, David R. Karger, Hariharan Rahul, and Mikkel Thorup. An experimental study of polylogarithmic, fully dynamic, connectivity algorithms. *ACM J. Exp. Algorithmics*, 6:4, 2001. doi:10.1145/945394.945398.
- 138 Hai Jin, Na Wang, Dongxiao Yu, Qiang-Sheng Hua, Xuanhua Shi, and Xia Xie. Core maintenance in dynamic graphs: A parallel approach based on matching. *IEEE Trans. Parallel Distributed Syst.*, 29(11):2416–2428, 2018. doi:10.1109/TPDS.2018.2835441.
- 139 Wenyu Jin and Xiaorui Sun. Fully dynamic c-edge connectivity in subpolynomial time. *CoRR*, abs/2004.07650, 2020. arXiv:2004.07650.

- 140 Hossein Jowhari and Mohammad Ghodsi. New streaming algorithms for counting triangles in graphs. In Lusheng Wang, editor, *Computing and Combinatorics, 11th Annual International Conference, COCOON 2005, Kunming, China, August 16-29, 2005, Proc.*, volume 3595 of *Lecture Notes in Computer Science*, pages 710–716. Springer, 2005. doi:10.1007/11533719_72.
- 141 Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in poly-logarithmic worst case time. In Sanjeev Khanna, editor, *Proc. of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1131–1142. SIAM, 2013. doi:10.1137/1.9781611973105.81.
- 142 Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Counting triangles under updates in worst-case optimal time. In Pablo Barceló and Marco Calautti, editors, *22nd International Conference on Database Theory, ICDT 2019, March 26-28, 2019, Lisbon, Portugal*, volume 127 of *LIPICs*, pages 4:1–4:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.ICDT.2019.4.
- 143 Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Maintaining triangle queries under updates. *ACM Trans. Database Syst.*, 45(3):11:1–11:46, 2020. doi:10.1145/3396375.
- 144 Miray Kas, Kathleen M. Carley, and L. Richard Carley. Incremental closeness centrality for dynamically changing social networks. In Jon G. Rokne and Christos Faloutsos, editors, *Advances in Social Networks Analysis and Mining 2013, ASONAM '13, Niagara, ON, Canada - August 25 - 29, 2013*, pages 1250–1258. ACM, 2013. doi:10.1145/2492517.2500270.
- 145 Shahbaz Khan. Near optimal parallel algorithms for dynamic DFS in undirected graphs. *ACM Trans. Parallel Comput.*, 6(3):18:1–18:33, 2019. doi:10.1145/3364212.
- 146 Tim Kiefer, Dirk Habich, and Wolfgang Lehner. Penalized graph partitioning for static and dynamic load balancing. In Pierre-François Dutot and Denis Trystram, editors, *Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proc.*, volume 9833 of *Lecture Notes in Computer Science*, pages 146–158. Springer, 2016. doi:10.1007/978-3-319-43659-3_11.
- 147 Valerie King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 81–91. IEEE Computer Society, 1999. doi:10.1109/SFFCS.1999.814580.
- 148 Valerie King and Mikkel Thorup. A space saving trick for directed dynamic transitive closure and shortest path algorithms. In Jie Wang, editor, *Computing and Combinatorics, 7th Annual International Conference, COCOON 2001, Guilin, China, August 20-23, 2001, Proc.*, volume 2108 of *Lecture Notes in Computer Science*, pages 268–277. Springer, 2001. doi:10.1007/3-540-44679-6_30.
- 149 Pushmeet Kohli and Philip H. S. Torr. Dynamic graph cuts for efficient inference in markov random fields. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29(12):2079–2088, 2007. doi:10.1109/TPAMI.2007.1128.
- 150 Pushmeet Kohli and Philip H. S. Torr. Dynamic graph cuts and their applications in computer vision. In Roberto Cipolla, Sebastiano Battiato, and Giovanni Maria Farinella, editors, *Computer Vision: Detection, Recognition and Reconstruction*, volume 285 of *Studies in Computational Intelligence*, pages 51–108. Springer, 2010. doi:10.1007/978-3-642-12848-6_3.
- 151 Nicolas Kourtellis, Gianmarco De Francisci Morales, and Francesco Bonchi. Scalable online betweenness centrality in evolving graphs. *IEEE Trans. Knowl. Data Eng.*, 27(9):2494–2506, 2015. doi:10.1109/TKDE.2015.2419666.
- 152 Nicolas Kourtellis, Gianmarco De Francisci Morales, and Francesco Bonchi. Scalable online betweenness centrality in evolving graphs. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 1580–1581. IEEE Computer Society, 2016. doi:10.1109/ICDE.2016.7498421.

- 153 Ioannis Krommidas and Christos D. Zaroliagis. An experimental study of algorithms for fully dynamic transitive closure. *ACM J. Exp. Algorithmics*, 12:1.6:1–1.6:22, 2008. doi:10.1145/1227161.1370597.
- 154 S. Kumar and P. Gupta. An incremental algorithm for the maximum flow problem. *J. Math. Model. Algorithms*, 2(1):1–16, 2003. doi:10.1023/A:1023607406540.
- 155 Min-Joong Lee, Sunghee Choi, and Chin-Wan Chung. Efficient algorithms for updating betweenness centrality in fully dynamic graphs. *Inf. Sci.*, 326:278–296, 2016. doi:10.1016/j.ins.2015.07.053.
- 156 Min-Joong Lee, Jungmin Lee, Jaimie Yejean Park, Ryan Hyun Choi, and Chin-Wan Chung. QUBE: a quick algorithm for updating betweenness centrality. In Alain Mille, Fabien L. Gandon, Jacques Misselis, Michael Rabinovich, and Steffen Staab, editors, *Proc. of the 21st World Wide Web Conference 2012, WWW 2012, Lyon, France, April 16-20, 2012*, pages 351–360. ACM, 2012. doi:10.1145/2187836.2187884.
- 157 Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. Efficient core maintenance in large dynamic graphs. *IEEE Trans. Knowl. Data Eng.*, 26(10):2453–2465, 2014. doi:10.1109/TKDE.2013.158.
- 158 Yongsub Lim, Minsoo Jung, and U Kang. Memory-efficient and accurate sampling for counting local triangles in graph streams: From simple to multigraphs. *ACM Trans. Knowl. Discov. Data*, 12(1):4:1–4:28, 2018. doi:10.1145/3022186.
- 159 Yongsub Lim and U Kang. MASCOT: memory-efficient and accurate sampling for counting local triangles in graph streams. In Longbing Cao, Chengqi Zhang, Thorsten Joachims, Geoffrey I. Webb, Dragos D. Margineantu, and Graham Williams, editors, *Proc. of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015*, pages 685–694. ACM, 2015. doi:10.1145/2783258.2783285.
- 160 Paul Liu, Austin R. Benson, and Moses Charikar. Sampling methods for counting temporal motifs. In J. Shane Culpepper, Alistair Moffat, Paul N. Bennett, and Kristina Lerman, editors, *Proc. of the Twelfth ACM International Conference on Web Search and Data Mining, WSDM 2019, Melbourne, VIC, Australia, February 11-15, 2019*, pages 294–302. ACM, 2019. doi:10.1145/3289600.3290988.
- 161 Shangqi Lu and Yufei Tao. Towards optimal dynamic indexes for approximate (and exact) triangle counting. In Ke Yi and Zhewei Wei, editors, *24th International Conference on Database Theory, ICDT 2021, March 23-26, 2021, Nicosia, Cyprus*, volume 186 of *LIPICs*, pages 6:1–6:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.ICDT.2021.6.
- 162 Amgad Madkour, Walid G Aref, Faizan Ur Rehman, Mohamed Abdur Rahman, and Saleh Basalamah. A survey of shortest-path algorithms. *arXiv preprint*, 2017. arXiv:1705.02044.
- 163 Devavret Makkar, David A. Bader, and Oded Green. Exact and parallel triangle counting in dynamic graphs. In *24th IEEE International Conference on High Performance Computing, HiPC 2017, Jaipur, India, December 18-21, 2017*, pages 2–12. IEEE Computer Society, 2017. doi:10.1109/HiPC.2017.00011.
- 164 Alberto Marchetti-Spaccamela, Umberto Nanni, and Hans Rohnert. Maintaining a topological order under edge insertions. *Inf. Process. Lett.*, 59(1):53–58, 1996. doi:10.1016/0020-0190(96)00075-0.
- 165 Henning Meyerhenke. Dynamic load balancing for parallel numerical simulations based on repartitioning with disturbed diffusion. In *15th International Conference on Parallel and Distributed Systems*, pages 150–157. IEEE, 2009. doi:10.1109/ICPADS.2009.114.
- 166 Henning Meyerhenke and Joachim Gehweiler. On dynamic graph partitioning and graph clustering using diffusion. In Giuseppe F. Italiano, David S. Johnson, Petra Mutzel, and Peter Sanders, editors, *Algorithm Engineering, 27.06. - 02.07.2010*, volume 10261 of *Dagstuhl Seminar Proc.* Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2010. URL: <http://drops.dagstuhl.de/opus/volltexte/2010/2798/>.

- 167 Kurt T Miller and Tina Eliassi-Rad. Continuous time group discovery in dynamic graphs. In *Notes of the 2009 NIPS Workshop on Analyzing Networks and Learning with Graphs, Whistler, BC, Canada, 2009*.
- 168 Peter Bro Miltersen, Sairam Subramanian, Jeffrey Scott Vitter, and Roberto Tamassia. Complexity models for incremental computation. *Theor. Comput. Sci.*, 130(1):203–236, 1994. doi:10.1016/0304-3975(94)90159-7.
- 169 Daniele Miorandi and Francesco De Pellegrini. K-shell decomposition for dynamic complex networks. In Lavy Libman Ariel Orda, Nidhi Hegde, editor, *8th International Symposium on Modeling and Optimization in Mobile, Ad-Hoc and Wireless Networks (WiOpt 2010), May 31 - June 4, 2010, University of Avignon, Avignon, France*, pages 488–496. IEEE, 2010. URL: <http://ieeexplore.ieee.org/document/5520231/>.
- 170 Sudip Misra and B. John Oommen. Dynamic algorithms for the shortest path routing problem: Learning automata-based solutions. *IEEE Trans. Syst. Man Cybern. Part B*, 35(6):1179–1192, 2005. doi:10.1109/TSMCB.2005.850180.
- 171 Kingshuk Mukherjee, Md Mahmudul Hasan, Christina Boucher, and Tamer Kahveci. Counting motifs in dynamic networks. *BMC Syst. Biol.*, 12(1):1–12, 2018. doi:10.1186/s12918-018-0533-6.
- 172 Kengo Nakamura and Kunihiro Sadakane. Space-efficient fully dynamic DFS in undirected graphs. *Algorithms*, 12(3):52, 2019. doi:10.3390/a12030052.
- 173 Paolo Narváez, Kai-Yeung Siu, and Hong-Yi Tzeng. New dynamic algorithms for shortest path tree computation. *IEEE/ACM Trans. Netw.*, 8(6):734–746, 2000. doi:10.1109/90.893870.
- 174 Paolo Narváez, Kai-Yeung Siu, and Hong-Yi Tzeng. New dynamic SPT algorithm based on a ball-and-string model. *IEEE/ACM Trans. Netw.*, 9(6):706–718, 2001. doi:10.1109/90.974525.
- 175 Meghana Nasre, Matteo Pontecorvi, and Vijaya Ramachandran. Betweenness centrality - incremental and faster. In Erzsébet Csuhaj-Varjú, Martin Dietzfelbinger, and Zoltán Ésik, editors, *Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part II*, volume 8635 of *Lecture Notes in Computer Science*, pages 577–588. Springer, 2014. doi:10.1007/978-3-662-44465-8_49.
- 176 Eisha Nathan and David A. Bader. Approximating personalized katz centrality in dynamic graphs. In Roman Wyrzykowski, Jack J. Dongarra, Ewa Deelman, and Konrad Karczewski, editors, *Parallel Processing and Applied Mathematics - 12th International Conference, PPAM 2017, Lublin, Poland, September 10-13, 2017, Revised Selected Papers, Part I*, volume 10777 of *Lecture Notes in Computer Science*, pages 290–302. Springer, 2017. doi:10.1007/978-3-319-78024-5_26.
- 177 Eisha Nathan and David A. Bader. A dynamic algorithm for updating katz centrality in graphs. In Jana Diesner, Elena Ferrari, and Guandong Xu, editors, *Proc. of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017, Sydney, Australia, July 31 - August 03, 2017*, pages 149–154. ACM, 2017. doi:10.1145/3110025.3110034.
- 178 Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. *ACM Trans. Algorithms*, 12(1):7:1–7:15, 2016. doi:10.1145/2700206.
- 179 M. EJ Newman and M. Girvan. Finding and Evaluating Community Structure in Networks. *Physical review E*, 69(2):026113, 2004. doi:10.1103/PhysRevE.69.026113.
- 180 Daniel Nicoara, Shahin Kamali, Khuzaima Daudjee, and Lei Chen. Hermes: Dynamic partitioning for distributed social network graph databases. In *EDBT*, pages 25–36, 2015. doi:10.5441/002/edbt.2015.04.
- 181 Krzysztof Onak and Ronitt Rubinfeld. Maintaining a large matching and a small vertex cover. In *STOC*, pages 457–464, 2010. doi:10.1145/1806689.1806753.
- 182 James B. Orlin. Max flows in $o(nm)$ time, or better. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 765–774. ACM, 2013. doi:10.1145/2488608.2488705.

- 183 Mihai Patrascu. Towards polynomial lower bounds for dynamic problems. In *Proc. of the 42nd ACM Symposium on Theory of Computing, STOC*, pages 603–610. ACM, 2010. doi:10.1145/1806689.1806772.
- 184 Mihai Patrascu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.*, 35(4):932–963, 2006. doi:10.1137/S0097539705447256.
- 185 A. Pavan, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Counting and sampling triangles from a graph stream. *Proc. VLDB Endow.*, 6(14):1870–1881, 2013. doi:10.14778/2556549.2556569.
- 186 D. Pearce and P. Kelly. A batch algorithm for maintaining a topological order. In *ACSC*, 2010. doi:10.5555/1862199.1862208.
- 187 David J. Pearce and Paul H. J. Kelly. A dynamic algorithm for topologically sorting directed acyclic graphs. In Celso C. Ribeiro and Simone L. Martins, editors, *Experimental and Efficient Algorithms, Third International Workshop, WEA 2004, Angra dos Reis, Brazil, May 25-28, 2004, Proc.*, volume 3059 of *Lecture Notes in Computer Science*, pages 383–398. Springer, 2004. doi:10.1007/978-3-540-24838-5_29.
- 188 David J. Pearce and Paul H. J. Kelly. A dynamic topological sort algorithm for directed acyclic graphs. *ACM J. Exp. Algorithmics*, 11, 2006. doi:10.1145/1187436.1210590.
- 189 David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Online cycle detection and difference propagation: Applications to pointer analysis. *Softw. Qual. J.*, 12(4):311–337, 2004. doi:10.1023/B:SQJ0.0000039791.93071.a2.
- 190 Reynaldo Gil Pons. Space efficient incremental betweenness algorithm for directed graphs. In Rubén Vera-Rodríguez, Julian Fierrez, and Aythami Morales, editors, *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications - 23rd Iberoamerican Congress, CIARP 2018, Madrid, Spain, November 19-22, 2018, Proc.*, volume 11401 of *Lecture Notes in Computer Science*, pages 262–270. Springer, 2018. doi:10.1007/978-3-030-13469-3_31.
- 191 Matteo Pontecorvi and Vijaya Ramachandran. Fully dynamic betweenness centrality. In Khaled M. Elbassioni and Kazuhisa Makino, editors, *Algorithms and Computation - 26th International Symposium, ISAAC 2015, Nagoya, Japan, December 9-11, 2015, Proceedings*, volume 9472 of *Lecture Notes in Computer Science*, pages 331–342. Springer, 2015. doi:10.1007/978-3-662-48971-0_29.
- 192 K. (Lynn) Putman, Hanjo D. Boekhout, and Frank W. Takes. Fast incremental computation of harmonic closeness centrality in directed weighted networks. In Francesca Spezzano, Wei Chen, and Xiaokui Xiao, editors, *ASONAM '19: International Conference on Advances in Social Networks Analysis and Mining, Vancouver, British Columbia, Canada, 27-30 August, 2019*, pages 1018–1025. ACM, 2019. doi:10.1145/3341161.3344829.
- 193 G Ramalingam and Thomas Reps. On the computational complexity of incremental algorithms. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1991.
- 194 G. Ramalingam and Thomas W. Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, 21(2):267–305, 1996. doi:10.1006/jagm.1996.0046.
- 195 G. Ramalingam and Thomas W. Reps. On the computational complexity of dynamic graph problems. *Theor. Comput. Sci.*, 158(1&2):233–277, 1996. doi:10.1016/0304-3975(95)00079-8.
- 196 Celso C. Ribeiro and Rodrigo F. Toso. Experimental analysis of algorithms for updating minimum spanning trees on graphs subject to changes on edge weights. In Camil Demetrescu, editor, *Experimental Algorithms, 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007, Proc.*, volume 4525 of *Lecture Notes in Computer Science*, pages 393–405. Springer, 2007. doi:10.1007/978-3-540-72845-0_30.
- 197 Matteo Riondato and Evgenios M. Kornaropoulos. Fast approximation of betweenness centrality through sampling. In Ben Carterette, Fernando Diaz, Carlos Castillo, and Donald Metzler, editors, *Seventh ACM International Conference on Web Search and Data Mining, WSDM 2014, New York, NY, USA, February 24-28, 2014*, pages 413–422. ACM, 2014. doi:10.1145/2556195.2556224.

- 198 Liam Roditty. A faster and simpler fully dynamic transitive closure. *ACM Trans. Algorithms*, 4(1):6:1–6:16, 2008. doi:10.1145/1328911.1328917.
- 199 Liam Roditty and Uri Zwick. On dynamic shortest paths problems. *Algorithmica*, 61(2):389–401, 2011. doi:10.1007/s00453-010-9401-5.
- 200 Liam Roditty and Uri Zwick. Dynamic approximate all-pairs shortest paths in undirected graphs. *SIAM J. Comput.*, 41(3):670–683, 2012. doi:10.1137/090776573.
- 201 Liam Roditty and Uri Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. *SIAM Journal on Computing*, 45(3):712–733, 2016. doi:10.1137/13093618X.
- 202 Giulio Rossetti and Rémy Cazabet. Community discovery in dynamic networks: A survey. *ACM Comput. Surv.*, 51(2):35:1–35:37, 2018. doi:10.1145/3172867.
- 203 Tiberiu Rotaru and Hans-Heinrich Nägeli. Dynamic load balancing by diffusion in heterogeneous systems. *Journal of Parallel and Distributed Computing*, 64(4):481–497, 2004. doi:10.1016/j.jpdc.2004.02.001.
- 204 Chayma Sakouhi, Sabeur Aridhi, Alessio Guerrieri, Salma Sassi, and Alberto Montresor. Dynamicdfep: A distributed edge partitioning approach for large dynamic graphs. In Evan Desai, Bipin C. Desai, Motomichi Toyama, and Jorge Bernardino, editors, *Proc. of the 20th International Database Engineering & Applications Symposium, IDEAS 2016, Montreal, QC, Canada, July 11-13, 2016*, pages 142–147. ACM, 2016. doi:10.1145/2938503.2938506.
- 205 Piotr Sankowski. Dynamic transitive closure via dynamic matrix inverse (extended abstract). In *45th Symposium on Foundations of Computer Science (FOCS 2004), 17-19 October 2004, Rome, Italy, Proc.*, pages 509–517. IEEE Computer Society, 2004. doi:10.1109/FOCS.2004.25.
- 206 Piotr Sankowski. Subquadratic algorithm for dynamic shortest distances. In Lusheng Wang, editor, *Computing and Combinatorics, 11th Annual International Conference, COCOON 2005, Kunming, China, August 16-29, 2005, Proceedings*, volume 3595 of *Lecture Notes in Computer Science*, pages 461–470. Springer, 2005. doi:10.1007/11533719_47.
- 207 Piotr Sankowski. Faster dynamic matchings and vertex connectivity. In *SODA*, pages 118–126, 2007. doi:10.1145/1283383.1283397.
- 208 Ahmet Erdem Sariyüce, Bugra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. Incremental k-core decomposition: algorithms and evaluation. *VLDB J.*, 25(3):425–447, 2016. doi:10.1007/s00778-016-0423-8.
- 209 Ahmet Erdem Sariyüce, Kamer Kaya, Erik Saule, and Ümit V. Çatalyürek. Incremental algorithms for closeness centrality. In Xiaohua Hu, Tsau Young Lin, Vijay V. Raghavan, Benjamin W. Wah, Ricardo Baeza-Yates, Geoffrey C. Fox, Cyrus Shahabi, Matthew Smith, Qiang Yang, Rayid Ghani, Wei Fan, Ronny Lempel, and Raghunath Nambiar, editors, *Proc. of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*, pages 487–492. IEEE Computer Society, 2013. doi:10.1109/BigData.2013.6691611.
- 210 Ahmet Erdem Sariyüce, Erik Saule, Kamer Kaya, and Ümit V. Çatalyürek. STREAMER: A distributed framework for incremental closeness centrality computation. In *2013 IEEE International Conference on Cluster Computing, CLUSTER 2013, Indianapolis, IN, USA, September 23-27, 2013*, pages 1–8. IEEE Computer Society, 2013. doi:10.1109/CLUSTER.2013.6702680.
- 211 Ahmet Erdem Sariyüce, Erik Saule, Kamer Kaya, and Ümit V. Çatalyürek. Incremental closeness centrality in distributed memory. *Parallel Comput.*, 47:3–18, 2015. doi:10.1016/j.parco.2015.01.003.
- 212 Benjamin Schiller, Sven Jager, Kay Hamacher, and Thorsten Strufe. Stream - A stream-based algorithm for counting motifs in dynamic graphs. In Adrian-Horia Dediu, Francisco Hernández Quiroz, Carlos Martín-Vide, and David A. Rosenblueth, editors, *Algorithms for Computational Biology - Second International Conference, AICoB 2015, Mexico City, Mexico, August 4-5, 2015, Proc.*, volume 9199 of *Lecture Notes in Computer Science*, pages 53–67. Springer, 2015. doi:10.1007/978-3-319-21233-3_5.

- 213 Kirk Schloegel, George Karypis, and Vipin Kumar. Multilevel diffusion schemes for re-partitioning of adaptive meshes. *J. Parallel Distributed Comput.*, 47(2):109–124, 1997. doi:10.1006/jpdc.1997.1410.
- 214 Kirk Schloegel, George Karypis, and Vipin Kumar. A unified algorithm for load-balancing adaptive scientific simulations. In Jed Donnelley, editor, *Proc. Supercomputing 2000, November 4-10, 2000, Dallas, Texas, USA. IEEE Computer Society, CD-ROM*, page 59. IEEE Computer Society, 2000. doi:10.1109/SC.2000.10035.
- 215 Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurr. Comput. Pract. Exp.*, 14(3):219–240, 2002. doi:10.1002/cpe.605.
- 216 Dominik Schultes and Peter Sanders. Dynamic highway-node routing. In Camil Demetrescu, editor, *Experimental Algorithms, 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007, Proc.*, volume 4525 of *Lecture Notes in Computer Science*, pages 66–79. Springer, 2007. doi:10.1007/978-3-540-72845-0_6.
- 217 Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. Graphin: An online high performance incremental graph processing framework. In *European Conference on Parallel Processing*, pages 319–333. Springer, 2016. doi:10.1007/978-3-319-43659-3_24.
- 218 Neha Sengupta, Michael Hamann, and Dorothea Wagner. Benchmark generator for dynamic overlapping communities in networks. In Vijay Raghavan, Srinivas Aluru, George Karypis, Lucio Miele, and Xindong Wu, editors, *2017 IEEE International Conference on Data Mining, ICDM 2017, New Orleans, LA, USA, November 18-21, 2017*, pages 415–424. IEEE Computer Society, 2017. doi:10.1109/ICDM.2017.51.
- 219 Zhenzhen Shao, Na Guo, Yu Gu, Zhigang Wang, Fangfang Li, and Ge Yu. Efficient closeness centrality computation for dynamic graphs. In Yunmook Nah, Bin Cui, Sang-Won Lee, Jeffrey Xu Yu, Yang-Sae Moon, and Steven Euijong Whang, editors, *Database Systems for Advanced Applications - 25th International Conference, DASFAA 2020, Jeju, South Korea, September 24-27, 2020, Proc., Part II*, volume 12113 of *Lecture Notes in Computer Science*, pages 534–550. Springer, 2020. doi:10.1007/978-3-030-59416-9_32.
- 220 Y. Shiloach and S. Even. An on-line edge-deletion problem. *Journal of the ACM*, 28(1):1–4, 1981. doi:10.1145/322234.322235.
- 221 Dhirendra Singh and Nilay Khare. Parallel batch dynamic single source shortest path algorithm and its implementation on GPU based machine. *Int. Arab J. Inf. Technol.*, 16(2):217–225, 2019. URL: http://iajit.org/index.php?option=com_content&task=blogcategory&id=137&Itemid=469.
- 222 Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. In *Proc. of the 13th Annual ACM Symposium on Theory of Computing, May 11-13, 1981, Milwaukee, Wisconsin, USA*, pages 114–122. ACM, 1981. doi:10.1145/800076.802464.
- 223 Shay Solomon. Fully dynamic maximal matching in constant update time. In *57th Symposium on Foundations of Computer Science FOCS*, pages 325–334, 2016. doi:10.1109/FOCS.2016.43.
- 224 Lorenzo De Stefani, Alessandro Epasto, Matteo Riondato, and Eli Upfal. Trièst: Counting local and global triangles in fully dynamic streams with fixed memory size. *ACM Trans. Knowl. Discov. Data*, 11(4):43:1–43:50, 2017. doi:10.1145/3059194.
- 225 Daniel Stubbs and Virginia Vassilevska Williams. Metatheorems for dynamic weighted matching. In Christos H. Papadimitriou, editor, *8th Innovations in Theoretical Computer Science Conference, ITCS 2017, January 9-11, 2017, Berkeley, CA, USA*, volume 67 of *LIPICs*, pages 58:1–58:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.ITCS.2017.58.
- 226 Bintao Sun, T.-H. Hubert Chan, and Mauro Sozio. Fully dynamic approximate k-core decomposition in hypergraphs. *ACM Trans. Knowl. Discov. Data*, 14(4), May 2020. doi:10.1145/3385416.

- 227 Robert Endre Tarjan and Renato Fonseca F. Werneck. Dynamic trees in practice. *ACM J. Exp. Algorithmics*, 14, 2009. doi:10.1145/1498698.1594231.
- 228 Mikkel Thorup. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In Torben Hagerup and Jyrki Katajainen, editors, *Algorithm Theory - SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory, Humlebaek, Denmark, July 8-10, 2004, Proceedings*, volume 3111 of *Lecture Notes in Computer Science*, pages 384–396. Springer, 2004. doi:10.1007/978-3-540-27810-8_33.
- 229 Mikkel Thorup. Worst-case update times for fully-dynamic all-pairs shortest paths. In Harold N. Gabow and Ronald Fagin, editors, *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 112–119. ACM, 2005. doi:10.1145/1060590.1060607.
- 230 Mikkel Thorup. Fully-dynamic min-cut. *Comb.*, 27(1):91–127, 2007. doi:10.1007/s00493-007-0045-2.
- 231 Mikkel Thorup and David R. Karger. Dynamic graph algorithms with applications. In Magnús M. Halldórsson, editor, *Algorithm Theory - SWAT 2000, 7th Scandinavian Workshop on Algorithm Theory, Bergen, Norway, July 5-7, 2000, Proceedings*, volume 1851 of *Lecture Notes in Computer Science*, pages 1–9. Springer, 2000. doi:10.1007/3-540-44985-X_1.
- 232 Thomas Tseng, Laxman Dhulipala, and Guy E. Blelloch. Batch-parallel euler tour trees. In Stephen G. Kobourov and Henning Meyerhenke, editors, *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2019, San Diego, CA, USA, January 7-8, 2019*, pages 92–106. SIAM, 2019. doi:10.1137/1.9781611975499.8.
- 233 Jan van den Brand and Danupon Nanongkai. Dynamic approximate shortest paths and beyond: Subquadratic and worst-case update time. In David Zuckerman, editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 436–455. IEEE Computer Society, 2019. doi:10.1109/FOCS.2019.00035.
- 234 Jan van den Brand, Danupon Nanongkai, and Thatchaphol Saranurak. Dynamic matrix inverse: Improved algorithms and matching conditional lower bounds. In David Zuckerman, editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 456–480. IEEE Computer Society, 2019. doi:10.1109/FOCS.2019.00036.
- 235 Alexander van der Grinten, Elisabetta Bergamini, Oded Green, David A. Bader, and Henning Meyerhenke. Scalable katz ranking computation in large static and dynamic graphs. In Yossi Azar, Hannah Bast, and Grzegorz Herman, editors, *26th Annual European Symposium on Algorithms, ESA 2018, August 20-22, 2018, Helsinki, Finland*, volume 112 of *LIPIcs*, pages 42:1–42:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.ESA.2018.42.
- 236 Luis M. Vaquero, Félix Cuadrado, Dionysios Logothetis, and Claudio Martella. xdgp: A dynamic graph processing system with adaptive partitioning. *CoRR*, abs/1309.1049, 2013. arXiv:1309.1049.
- 237 Luis M. Vaquero, Félix Cuadrado, Dionysios Logothetis, and Claudio Martella. Adaptive partitioning for large-scale dynamic graphs. In *IEEE 34th International Conference on Distributed Computing Systems, ICDCS 2014, Madrid, Spain, June 30 - July 3, 2014*, pages 144–153. IEEE Computer Society, 2014. doi:10.1109/ICDCS.2014.23.
- 238 Tanmay Verma and Dhruv Batra. Maxflow revisited: An empirical comparison of maxflow algorithms for dense vision problems. In Richard Bowden, John P. Collomosse, and Krystian Mikolajczyk, editors, *British Machine Vision Conference, BMVC 2012, Surrey, UK, September 3-7, 2012*, pages 1–12. BMVA Press, 2012. doi:10.5244/C.26.61.
- 239 Dorothea Wagner, Thomas Willhalm, and Christos D. Zaroliagis. Geometric containers for efficient shortest-path computation. *ACM J. Exp. Algorithmics*, 10, 2005. doi:10.1145/1064546.1103378.

- 240 Chris Walshaw, Mark Cross, and Martin G. Everett. Parallel dynamic graph partitioning for adaptive unstructured meshes. *J. Parallel Distributed Comput.*, 47(2):102–108, 1997. doi:10.1006/jpdc.1997.1407.
- 241 Jingjing Wang, Yanhao Wang, Wenjun Jiang, Yuchen Li, and Kian-Lee Tan. Efficient sampling algorithms for approximate temporal motif counting. In Mathieu d’Aquin, Stefan Dietze, Claudia Hauff, Edward Curry, and Philippe Cudré-Mauroux, editors, *CIKM ’20: The 29th ACM International Conference on Information and Knowledge Management, Virtual Event, Ireland, October 19-23, 2020*, pages 1505–1514. ACM, 2020. doi:10.1145/3340531.3411862.
- 242 Na Wang, Dongxiao Yu, Hai Jin, Chen Qian, Xia Xie, and Qiang-Sheng Hua. Parallel algorithm for core maintenance in dynamic graphs. In Kisung Lee and Ling Liu, editors, *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*, pages 2366–2371. IEEE Computer Society, 2017. doi:10.1109/ICDCS.2017.288.
- 243 Stefan Weigert, Matti Hiltunen, and Christof Fetzer. Mining large distributed log data in near real time. In *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, pages 1–8. Association for Computing Machinery, 2011. doi:10.1145/2038633.2038638.
- 244 Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. faimgraph: high performance management of fully-dynamic graphs under tight memory constraints on the GPU. In *Proc. of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*, pages 60:1–60:13. IEEE / ACM, 2018. URL: <http://dl.acm.org/citation.cfm?id=3291736>.
- 245 Ning Xu, Lei Chen, and Bin Cui. Loggp: a log-based dynamic graph partitioning method. *Proc. of the VLDB Endowment*, 7(14):1917–1928, 2014. doi:10.14778/2733085.2733097.
- 246 Bohua Yang, Dong Wen, Lu Qin, Ying Zhang, Xubo Wang, and Xuemin Lin. Fully dynamic depth-first search in directed graphs. *Proc. VLDB Endow.*, 13(2):142–154, 2019. doi:10.14778/3364324.3364329.
- 247 Chia-Chen Yen, Mi-Yen Yeh, and Ming-Syan Chen. An efficient approach to updating closeness centrality and average path length in dynamic networks. In Hui Xiong, George Karypis, Bhavani M. Thuraisingham, Diane J. Cook, and Xindong Wu, editors, *2013 IEEE 13th International Conference on Data Mining, Dallas, TX, USA, December 7-10, 2013*, pages 867–876. IEEE Computer Society, 2013. doi:10.1109/ICDM.2013.135.
- 248 Anita Zakrzewska and David A. Bader. Fast incremental community detection on dynamic graphs. In Roman Wyrzykowski, Ewa Deelman, Jack Dongarra, Konrad Karczewski, Jacek Kitowski, and Kazimierz Wiatr, editors, *Parallel Processing and Applied Mathematics*, pages 207–217, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-32149-3_20.
- 249 Christos D Zaroliagis. Implementations and experimental studies of dynamic graph algorithms. In *Experimental algorithmics*, pages 229–278. Springer, 2002. doi:10.1007/3-540-36383-1_11.
- 250 Yikai Zhang, Jeffrey Xu Yu, Ying Zhang, and Lu Qin. A fast order-based approach for core maintenance. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 337–348. IEEE Computer Society, 2017. doi:10.1109/ICDE.2017.93.
- 251 Weiguo Zheng, Chengzhi Piao, Hong Cheng, and Jeffrey Xu Yu. Computing a near-maximum independent set in dynamic graphs. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 76–87. IEEE, 2019. doi:10.1109/ICDE.2019.00016.
- 252 Weiguo Zheng, Qichen Wang, Jeffrey Xu Yu, Hong Cheng, and Lei Zou. Efficient computation of a near-maximum independent set over evolving graphs. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 869–880. IEEE Computer Society, 2018. doi:10.1109/ICDE.2018.00083.

- 253 Lei Zhu, Shaoning Pang, Abdolhossein Sarrafzadeh, Tao Ban, and Daisuke Inoue. Incremental and decremental max-flow for online semi-supervised learning. *IEEE Trans. Knowl. Data Eng.*, 28(8):2115–2127, 2016. doi:10.1109/TKDE.2016.2550042.
- 254 Di Zhuang, Morris J Chang, and Mingchen Li. Dynamo: Dynamic community detection by incrementally maximizing modularity. *IEEE Transactions on Knowledge and Data Engineering*, 2019. doi:10.1109/TKDE.2019.2951419.

Algorithmic Problems on Temporal Graphs

Paul G. Spirakis  

Department of Computer Science, University of Liverpool, UK

Computer Engineering & Informatics Department, University of Patras, Greece

Abstract

Research on Temporal Graphs has expanded in the last few years. Most of the results till now, address problems related to the notion of Temporal Paths (and Temporal Connectivity). In this talk, we focus, instead, on problems whose main topic is not on Temporal Paths. In particular, we will discuss Temporal Vertex Covers, the notion of Temporal Transitivity, and also issues and models of stochastic temporal graphs. We believe that several algorithmic graph problems, not directly related to paths, can be raised in the temporal domain. This may motivate new research towards lifting more topics of algorithmic graph theory to the temporal case.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Mathematics of computing → Discrete mathematics

Keywords and phrases Temporal graph, stochastic temporal graph, vertex cover, temporal transitivity

Digital Object Identifier 10.4230/LIPIcs.SAND.2022.2

Category Invited Talk



© Paul G. Spirakis;

licensed under Creative Commons License CC-BY 4.0

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).

Editors: James Aspnes and Othon Michail; Article No. 2; pp. 2:1–2:1

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Networks, Dynamics, Algorithms, and Learning

Roger Wattenhofer  

ETH Zürich, Switzerland

Abstract

Networks are notoriously difficult to understand, and adding dynamics does not help. Can the current wonder weapon of computation (yes, machine learning) come to the rescue? Unfortunately, learning with networks is generally not well understood. “Neural network networks” (better and less confusingly known as graph neural networks) can learn simple graph patterns, but they are a far cry from their impressive machine learning cousins in the image- or the game-domain. In my opinion, the most astonishing graph neural networks are in fact dealing with dynamic networks: They simulate sand (the granular material, not the symposium) quite naturally. In my talk, I will discuss and compare different computational objects and paradigms: networks, dynamics, algorithms, and learning. What are the differences? And what can they learn from each other? In the technical part of the talk, I will present DropGNN, our new algorithm-inspired approach for handling graph neural networks. But mostly I will vent about misunderstandings and mistakes, and I will propose open questions, and new research directions. DropGNN is joint work with Pál András Papp, Karolis Martinkus, and Lukas Faber, published at NeurIPS, December 2021.

2012 ACM Subject Classification Computing methodologies → Neural networks; Theory of computation → Distributed computing models

Keywords and phrases graph neural networks

Digital Object Identifier 10.4230/LIPIcs.SAND.2022.3

Category Invited Talk



© Roger Wattenhofer;

licensed under Creative Commons License CC-BY 4.0

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).

Editors: James Aspnes and Othon Michail; Article No. 3; pp. 3:1–3:1

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Atomic Splittable Flow Over Time Games

Antonia Adamik 

Technische Universität Berlin, Germany

Leon Sering  

ETH Zürich, Switzerland

Abstract

In an atomic splittable flow over time game, finitely many players route flow dynamically through a network, in which edges are equipped with transit times, specifying the traversing time, and with capacities, restricting flow rates. Infinitesimally small flow particles controlled by the same player arrive at a constant rate at the player's origin and the player's goal is to maximize the flow volume that arrives at the player's destination within a given time horizon. Here, the flow dynamics are described by the deterministic queuing model, i.e., flow of different players merges perfectly, but excessive flow has to wait in a queue in front of the bottle-neck. In order to determine Nash equilibria in such games, the main challenge is to consider suitable definitions for the players' strategies, which depend on the level of information the players receive throughout the game. For the most restricted version, in which the players receive no information on the network state at all, we can show that there is no Nash equilibrium in general, not even for networks with only two edges. However, if the current edge congestions are provided over time, the players can adapt their route choices dynamically. We show that a profile of those strategies always lead to a unique feasible flow over time. Hence, those atomic splittable flow over time games are well-defined. For parallel-edge networks Nash equilibria exists and the total flow arriving in time equals the value of a maximum flow over time leading to a price of anarchy of 1.

2012 ACM Subject Classification Theory of computation → Network flows; Theory of computation → Network games; Mathematics of computing → Network flows; Theory of computation → Quality of equilibria

Keywords and phrases Flows Over Time, Deterministic Queuing, Atomic Splittable Games, Equilibria, Traffic, Cooperation

Digital Object Identifier 10.4230/LIPIcs.SAND.2022.4

Related Version *Full Version:* <https://arxiv.org/abs/2010.02148>

Funding *Leon Sering:* Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – The Berlin Mathematics Research Center MATH+ (EXC-2046/1, project ID: 390685689).

Acknowledgements We have considered atomic splittable flow over time games in different settings and under various assumptions in collaboration with several people. Unfortunately, most of these research directions were more challenging than expected and not as successful as the work at hand. Nonetheless, we want to thank Laura Vargas Koch, Veerle Timmermans, Björn Tauer, Tim Oosterwijk and Dario Frascaria for the excellent collaboration and inspiring discussions.

1 Introduction

In *static routing problems*, traffic is to be routed through a network at minimum total cost. The cost or traveling time on each edge depends on its congestion. However, the assumption that an optimal routing might be implemented by some superordinate authority is not realistic in many settings. More likely, each network participant selfishly chooses a path in order to minimize their own traveling time. In general, the lack of coordination causes a higher total traveling time. To quantify this decrease in performance, the total traveling time



© Antonia Adamik and Leon Sering;

licensed under Creative Commons License CC-BY 4.0

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).

Editors: James Aspnes and Othon Michail; Article No. 4; pp. 4:1–4:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of a *Wardrop equilibrium* [45] is compared to the total traveling time of the *system optimum*. The ratio between a worst equilibrium and the system optimum is the *price of anarchy* [38]. This basic model can be extended in several ways. In this research work we want to focus on two aspects.

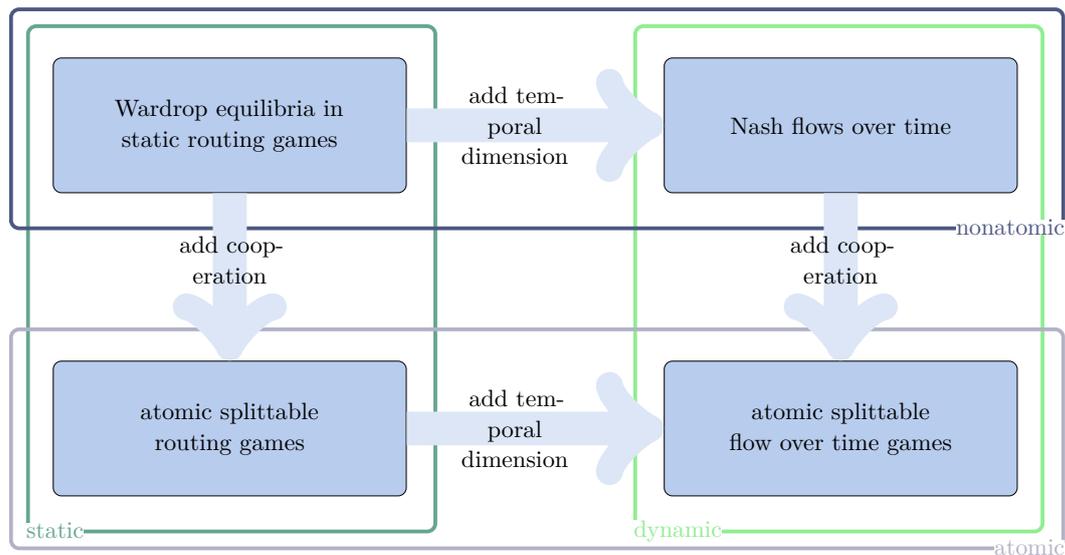
The first aspect is the temporal dimension. Vehicles in real traffic need time to move from the origin to the destination and the traveling time increases with the degree of congestion, which varies over time. In other words, the traffic flow does not traverse the network instantaneously, but progresses at a certain pace. In addition, the effects of a routing decision in one part of the network take some time to spread across the network as a whole. In order to mathematically model this, we add a time component, transforming static flows into *flows over time*. Here, every infinitesimally small flow particle needs time to traverse the network and the flow rates of the edges are restricted by capacities. By assuming that each particle acts selfishly, we can consider dynamic equilibria, which are called *Nash flows over time* [29].

For the second aspect, note that in real-world traffic the activity of a single road user has in most cases a negligible impact on the performance of the system as a whole. Furthermore, the assumption of independent and selfish particles is not justifiable in all applications: Networks where participants control a flow of positive measure are not covered. For instance, in transportation networks freight units might not act selfishly; they are controlled by freight companies that each control a significant amount of traffic. This leads to the second aspect, cooperative behavior among groups of network participants. To integrate this into the mathematical model, flow particles are allowed to form coalitions. In so-called *atomic splittable routing games* we consider a finite number of *atomic* players (the coalitions), each controlling a positive amount of flow volume that has to be *routed* through the network but can be *split up* and divided over different routes.

In this paper we want to combine both aspects, as depicted in Figure 1. That means, in contrast to Nash flows over time, sets of particles form coalitions which will be represented by superordinate players. In contrast to atomic splittable routing games, flow is modeled by a flow over time and players' decisions might be adapted to new situations. This extension covers a greater variety of scenarios. For example, road traffic models in which most drivers are guided by navigation systems (e.g. Google Maps, TomTom, Here, Garmin) can be modeled by covering the strategic behavior of the firms: The decisions of a single driver do not have a big impact on the city's traffic, but Google Maps decisions do; and TomTom might actually want to react. The situation will even intensify in the future with the rise of autonomous driving, as the decision making process is shifted to the navigation systems. We would like to point out that – in contrast to previous models – the interests of navigation companies and the general public are assumably in line: The cooperation of the users of a navigation system could reduce the average driving time per company on the one hand and the driving time in general on the other hand. This would lead to lower energy consumption, and therefore, lower emissions of polluting substances.

It turned out to be surprisingly challenging to consider equilibria in atomic splittable flow over time games. For this reason the overall goal is to define a solid model on these dynamic games and to present some preliminary observations, as well as some non-trivial first results, which serve as a basis for further research.

Related work. Static network flows have been studied for quite a while. A lot of pioneer work is due to Ford and Fulkerson, who also were the first to introduce *flows over time* [15, 16]. They provided an efficient algorithm for a *maximum flow over time*, which sends the maximal flow volume from a source to a sink given a finite time horizon. Closely related, a *quickest flow*



■ **Figure 1** Relationship between equilibrium situations in static routing games and atomic splittable flow over time games.

minimizes the arrival time of the latest particle for a given flow volume. This can be achieved by combining the algorithm of Ford and Fulkerson with a binary search framework [6, 14]. For single-source and single-sink networks it is furthermore possible to construct a flow over time that is maximal for all time horizons (and quickest for all flow volumes) simultaneously. The existence of these so-called *earliest arrival flows* was shown by Gale in 1959 [17]. They can be computed algorithmically by using the successive shortest path algorithm in the residual networks [31]. For more details and further references to literature on optimization problems in the flow over time setting, we refer to the survey of Skutella [43].

Koch and Skutella [29] approach flows over time from a game theoretic perspective by introducing *Nash flows over time*. In their model, every infinitesimally small flow particle is considered to be a player aiming to reach the common destination as early as possible. As the flow rate entering an edge could exceed its capacity, they considered the *deterministic queuing model* [44], which causes the excess flow to wait in a queue in front of the bottle-neck. Existence of these *dynamic equilibria* were shown by Cominetti et al. [8]. Several other aspects, including uniqueness, continuity, long term behavior, multi-terminals, spillback and price of anarchy, were studied in recent years [3, 9, 10, 12, 27, 32, 34, 40, 41]; see [39] for an overview. A slightly different approach for user equilibria was presented by Graf et al. [18, 19, 20]. They use the same flow over time model, except that particles do not anticipate the future evolution of the flow, but instead choose quickest routes according to current waiting times. As these delays may be subject to change, each particle can adapt its route choice along the way.

Atomic splittable congestion games for static network flows can be described as Wardrop equilibria [45] with coalitions [25, 30]; see also the survey of Correa and Stier-Moses [13]. For these games, Nash equilibria always exist, which can be shown by standard fixed point techniques [35]. Altman et al. [1] showed that equilibria are unique if the delay functions are polynomials of degree less than 3. Regarding more general delay functions, Bhaskar et al. [4] showed that for two players a unique equilibrium exists if, and only if, the network is a *generalized series-parallel graph*. Harks and Timmermans [24] showed uniqueness of

equilibria when the players' strategy space has a bidirectional flow polymatroid structure. Roughgarden [36] showed that the inefficiency of a system decreases with an increasing degree of cooperation. He showed that the price of anarchy for classes of traveling time functions in the atomic case is bounded by the price of anarchy for the same class of functions in the nonatomic case. Further research on the price of anarchy in static atomic splittable games is due to Cominetti et al. [11], Harks [21], and Roughgarden and Schoppmann [37]. Computational-wise Cominetti et al. [11] showed that equilibria can be computed efficiently when the cost functions are affine and player-independent. Regarding player-specific affine costs, Harks and Timmermans [23] described a polynomial algorithm for parallel-edge networks, and Bhaskar and Lolakapuri [5] presented an exponential algorithm for general convex functions. Very recently, Klimm and Warode showed that the computation with player-specific affine costs is PPAD-complete for general networks [28].

We should also mention that the combination of cooperation and temporal dimension has been considered for discrete packet routing games; see Peis et al. [33]. Here, each player controls a finite amount of packets, which has to be routed through a network in discrete time steps. For more results on competitive packet routing models we refer to Hoefer et al. [26] (continuous-time packets model) and Harks et al. [22] (discrete-time packet model).

Contribution and overview. In Section 2, we introduce all notations and formally describe atomic splittable flow over time games for general networks. The players' strategies determine how much flow they assign to each edge for every point in time during the game depending on available information on the current state. We consider two very natural sets of information. The first consists solely of the current time. In Section 3 we show that this setting does not allow for a Nash equilibrium in general, not even in a network with only two parallel edges. This motivates to consider more complex information models. Hence, Section 4 is dedicated to the second set of information which additionally comprises the current congestion of the edges in form of the exit times. As the first main result we show that every strategy profile results in a unique feasible flow over time by formulating the conditions as initial value problem and applying the Picard-Lindelöf theorem. For parallel-edge networks we show that Nash equilibria always exist by explicitly stating a strategy profile. Furthermore, we prove for that setting that all Nash equilibria have the same objective equal to the system optimum (i.e., the value of a maximum flow over time) implying that the price of anarchy for those networks is 1. Finally, we suggest further areas of research in Section 5.

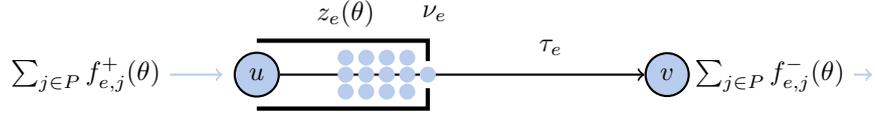
2 Atomic Splittable Flow Over Time Games

In this section, we are going to properly define atomic splittable flow over time games. The two main aspects are the multi-commodity flow dynamics (see [40]) and the players' strategies, which depend on the information received over time.

Game setting. A *network* consists of a directed graph $G = (V, E)$, where every edge $e \in E$ is equipped with a *transit time* $\tau_e > 0$ and a *capacity* $\nu_e > 0$. For a node $v \in V$ we denote the set of all incoming edges by δ_v^- and the set of outgoing edges by δ_v^+ .

For an *atomic splittable flow over time game* we consider a finite set of players P , each with an origin-destination pair s_j-t_j and a *supply rate* $d_j > 0$, as well as a time horizon $H > 0$. We assume that s_j can reach t_j within the network.

The flow of player j enters the network via node s_j at a rate of d_j from time 0 onwards. The goal is to maximize the cumulative flow volume reaching node t_j before the end of the game at time H .



■ **Figure 2** Representation of an edge in the deterministic queuing model: If more flow particles enter edge $e = uv$ within the total inflow rate $\sum_{j \in P} f_{e,j}^+(\theta)$ than its capacity ν_e allows to process, they build up a queue, whose current length is given by $z_e(\theta)$. Whenever the queue is non-empty at time θ the total outflow rate $\sum_{j \in P} f_{e,j}^-(\theta + \tau_e)$ at time $\theta + \tau_e$ equals the capacity ν_e .

Flow dynamics. In the deterministic queuing model the total inflow rate into an edge is bounded by the capacity. If the capacity is exceeded, a queue builds up in which all entering particles have to wait in line. Afterwards each particle needs τ_e time to traverse the edge before it can enter the next edge along the path; see Figure 2. The dynamics of this process are formally defined as follows:

A *flow over time* is described by a family of Lebesgue-integrable functions $f = (f^+, f^-) = (f_{e,j}^+, f_{e,j}^-)_{e \in E, j \in P}$, where $f_{e,j}^+, f_{e,j}^- : [0, H) \rightarrow \mathbb{R}_{\geq 0}$ denote the rate at which flow controlled by player j enters and leaves edge e . The flow rates summed over all players $\sum_{j \in P} f_{e,j}^+(\theta)$ and $\sum_{j \in P} f_{e,j}^-(\theta)$ are called *total in- and outflow rates*. The *cumulative in- and outflows*, i.e., the amount of player j 's flow that has entered or left an edge e up to time θ , is denoted by $F_{e,j}^+(\theta) = \int_0^\theta f_{e,j}^+(\vartheta) d\vartheta$ and $F_{e,j}^-(\theta) = \int_0^\theta f_{e,j}^-(\vartheta) d\vartheta$. Finally, f^+, f^-, F^+ and F^- denote the vectors of (cumulative) in- and outflows with one entry per edge-player-pair.

Such a family of functions f is a *flow over time* if the following two conditions hold for all $e \in E$ and $j \in P$:

Flow conservation is fulfilled for all $\theta \in [0, H)$:

$$\sum_{e \in \delta_v^+} f_{e,j}^+(\theta) - \sum_{e \in \delta_v^-} f_{e,j}^-(\theta) = \begin{cases} d_j & \text{for } v = s_j, \\ 0 & \text{for } v \in V \setminus \{s_j, t_j\}. \end{cases} \quad (1)$$

Non-deficit constraints are satisfied for all $\theta \in [0, H - \tau_e)$:

$$F_{e,j}^+(\theta) - F_{e,j}^-(\theta + \tau_e) \geq 0. \quad (2)$$

To track the net flow that is not yet processed and remains in the *queue*, we introduce $z_e(\theta)$ to denote the *queue length* at time θ . Formally, it is defined as $z_e(\theta) = \sum_{j \in P} F_{e,j}^+(\theta) - \sum_{j \in P} F_{e,j}^-(\theta + \tau_e)$ for $e \in E$. For a *feasible* flow over time we require that, whenever flow is waiting in the queue, the edge operates at capacity rate. In other words, for all $\theta \in [0, H - \tau_e)$ and $e \in E$, we require

$$\sum_{j \in P} f_{e,j}^-(\theta + \tau_e) = \begin{cases} \nu_e & \text{if } z_e(\theta) > 0, \\ \min \{ \sum_{j \in P} f_{e,j}^+(\theta), \nu_e \} & \text{else.} \end{cases} \quad (3)$$

The *waiting time* $q_e(\theta)$ experienced by a particle entering edge e at time θ is generally defined as the time needed to process the flow present in the queue when the particle enters it. In other words, it is the time span between its entrance to and its exit from the queue just before traversing the edge. That is

$$q_e(\theta) := \min \left\{ q \geq 0 \mid \int_\theta^{\theta+q} \sum_{j \in P} f_{e,j}^-(\vartheta + \tau_e) d\vartheta = z_e(\theta) \right\} = \frac{z_e(\theta)}{\nu_e}.$$

4:6 Atomic Splittable Flow Over Time Games

The *exit time* $T_e(\theta)$ of a particle entering an edge $e \in E$ at time θ is given by the sum of the entrance time θ , its waiting time in the queue $q_e(\theta)$ and the transit time τ_e . Hence, we have

$$T_e(\theta) := \theta + \frac{z_e(\theta)}{\nu_e} + \tau_e.$$

Since z_e can at most decrease at rate ν_e , it follows that $q'_e(\theta) \geq -1$ and $T'_e(\theta) \geq 0$, which induces that T_e is non-decreasing. Note that these derivatives exist for almost all θ due to Lebesgue's differentiation theorem.

Finally, in a feasible flow over time, the flow of different players should merge seamlessly. This means that, at any point in time, a player's share of the total outflow rate is equal to her share of the total inflow rate back at the time when the flow entered the edge. More precisely, we require

$$f_{e,j}^-(\theta) = \begin{cases} \frac{f_{e,j}^+(\phi)}{\sum_{j' \in P} f_{e,j'}^+(\phi)} \cdot \sum_{j' \in P} f_{e,j'}^-(\theta) & \text{if } \sum_{j' \in P} f_{e,j'}^+(\phi) > 0, \\ 0 & \text{else,} \end{cases} \quad (4)$$

for all $\theta \in [0, H)$ and $\phi := \max T_e^{-1}(\theta)$. Here, we set $f_{e,j}^+(\phi) := 0$ for $\phi < 0$. Note that ϕ denotes the edge-entrance times of all particles leaving the edge at time θ . Taking the maximum of $T_e^{-1}(\theta)$ simply ensures well-definedness which is required since T_e might not be strictly increasing.

To conclude we denote the set of all *feasible flows over time* by \mathcal{F} , i.e., all $f = (f^+, f^-)$ that satisfy (1), (2), (3) and (4). As the outflow rates are uniquely defined by the inflow rates, we refer to a feasible flow over time only by the corresponding inflow f^+ , and write $f^+ \in \mathcal{F}$.

Atomic splittable flow over time games. Let $\rho_j : \mathcal{F} \rightarrow \mathbb{R}$ be the function indicating player j 's *payoff* for a given $f \in \mathcal{F}$, which is to be maximized. In general, ρ can be set to various objective functions (e.g. arrival time of the player's latest particle or the average arrival time), but in this paper we will focus on the *maximum flow over time problem*. Each player wants to maximize her amount of flow routed from s_j to t_j before the end of the game at time H :

$$\rho_j(f) = \sum_{e \in \delta_{t_j}^-} F_{e,j}^-(H) - \sum_{e \in \delta_{t_j}^+} F_{e,j}^+(H).$$

We choose this maximum flow objective as it seems to be the most straight-forward payoff-function. It is conceivable though, that most results might transfer to quickest flow payoff-functions via a binary-search framework (cf. in non-competitive settings quickest flows are constructed from maximum flows over time via binary-search). But we leave this for future research.

The *strategy space* is a player's set of viable options in order to maximize her payoff. A single *strategy* is a complete instruction determining the player's inflow rates of all times and for all situations possibly occurring. Formally, the strategy space of player j is a set of functions

$$\Sigma_j = \left\{ g_j : \mathcal{I} \rightarrow [0, 1]^E \mid \begin{array}{l} g_{e,j} \text{ is Lebesgue-integrable for all } e \in E \text{ and} \\ \sum_{e \in \delta_v^+} g_{e,j}(I) = 1 \text{ for all } I \in \mathcal{I}, v \in V \setminus \{t_j\} \end{array} \right\},$$

where \mathcal{I} is the set of information available to the players. This set is not well-defined yet, but we will discuss this extensively, and in the end, we will consider two separate definitions for \mathcal{I} , one in Section 3 and one in Section 4. Informally, the set of information is used to delineate what defines a situation and how it is perceived by the players. The interpretation is as follows. For every information $I \in \mathcal{I}$, the value $g_{e,j}(I)$ determines which proportion of player j 's flow arriving at v is distributed onto the outgoing edges $e \in \delta_v^+$. We use proportions that sum up to 1 instead of the inflow rates, as this easily ensures that flow conservation is fulfilled at all times. Note that the received information I can depend on the current time θ and on the flow over time f itself. As we normally do not want the players to see the future, it should only depend on the flow over time up to time θ . In general it might be player dependent, hence, we write $I_j(\theta, f)$.

In order to turn a *strategy profile* $g = (g_j)_{j \in P} \in \times_{j \in P} \Sigma_j$ into a feasible flow over time f , we consider the following system of equations that has to be satisfied for all $\theta \in [0, H)$:

$$\begin{aligned} f_{e,j}^+(\theta) &= g_{e,j}(I_j(\theta, f)) \cdot \left(\sum_{e' \in \delta_{s_j}^-} f_{e',j}^-(\theta) + d_j \right) \quad \text{for } j \in P, e \in \delta_{s_j}^+, \\ f_{e,j}^+(\theta) &= g_{e,j}(I_j(\theta, f)) \cdot \sum_{e' \in \delta_u^-} f_{e',j}^-(\theta) \quad \text{for } j \in P, e = uv \in E \setminus (\delta_{s_j}^+ \cup \delta_{t_j}^+), \\ f_{e,j}^+(\theta) &= 0 \quad \text{for } j \in P, e \in \delta_{t_j}^+. \end{aligned} \tag{5}$$

Note that in order to keep it as simple as possible, we assume that flow of player j reaching t_j leaves the network immediately, hence, $f_{e,j}^+(\theta) = 0$ for all $e \in \delta_{t_j}^+$. This leads to a simpler payoff for all players j of $\rho_j(f) = \sum_{e \in \delta_{t_j}^-} F_{e,j}^-(H)$.

Since the inflow rates $f^+(\theta)$ might depend on the flow over time itself up to time θ , it is not guaranteed that this system of equations yields a feasible flow over time as solution.

To illustrate this issue assume for example a game with a single player in a network with two parallel edges e_1 and e_2 where e_1 has a tiny capacity. If the player's strategy is to send everything into e_1 as long as there is no waiting time on e_1 , and otherwise send everything into e_2 , this would not result in a feasible flow over time. To see this, suppose that the inflow into e_1 would be positive on a measurable set, which would immediately cause a positive waiting time on e_1 . Hence, the strategy says that no flow is sent into e_1 . On the other hand, if no flow is sent into e_1 at all there would not be any waiting time, leading again to a contradiction.

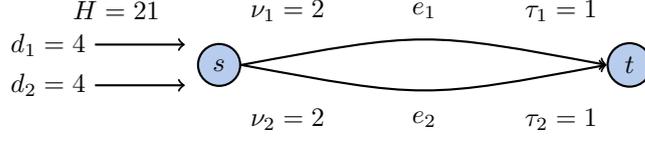
For that reason, the key challenge is to find a reasonable set of information \mathcal{I} and strategy spaces Σ_j , such that, on the one hand, there exists a unique (up to a null set) feasible flow over time satisfying (5) for every given strategy profile g , and on the other hand, a Nash equilibrium exists. Note that we only consider *pure* Nash equilibria.

In the following two sections we discuss two very natural sets of information.

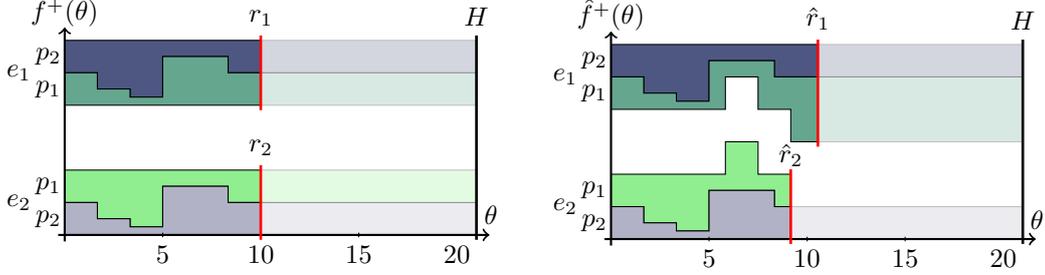
3 Temporal Information Only

First, we want to examine the simplest set of information, namely the current point in time only: We set $I_j(\theta, f) := \theta$ for all players $j \in P$. That means the players do not receive any information about the current state of the flow, but instead have to decide at the beginning of the game along which routes their flow particles are routed. In this case it is guaranteed that there exists a unique feasible flow over time that satisfies (5), as we can simply set $f_{e,j}^+(\theta)$ to the right sides of (5) (formally this also follows from Theorem 2). However, Nash equilibria do not exist in general, which is already true for very simple networks.

4:8 Atomic Splittable Flow Over Time Games



(a) A parallel network with two identical edges and two identical players.



(b) Step 1: p_1 mirrors p_2 's strategy. We have $r_1 = r_2 = 10$ and both queues build up from $\theta = 0$ onward. W.l.o.g., we have $f_{1,2}^+(\theta) \leq d_2 = 4$ for $\theta \in [10 - \varepsilon, 10)$.

(c) Step 2: p_1 shifts δ flow units from e_1 to e_2 causing $\hat{r}_1 > 10$ and $\hat{r}_2 < 10$. By pumping at maximum rate into e_1 after \hat{r}_2 her payoff is $\hat{\rho}_1 > 40 = \text{OPT} / 2$.

■ **Figure 3** Construction of a response strategy that always yields strictly more than half of the total optimum in a network of two parallel edges. Here, r_i and \hat{r}_i denote the time when the last particles arriving at t just in time, enter e_i .

► **Theorem 1.** *With temporal information only, there exists no Nash equilibrium in an atomic splittable flow over time game of two players p_1 and p_2 with identical supply rates $d_1, d_2 > 0$, on a network with two identical parallel edges e_1, e_2 from s ($= s_{p_1} = s_{p_2}$) to t ($= t_{p_1} = t_{p_2}$) with $\nu_{e_1} = \nu_{e_2} < d_1 = d_2$ and $\tau_{e_1} = \tau_{e_2} < H$.*

The key proof idea is the following. To every strategy of the competitor, a player can choose a response strategy that yields a payoff of strictly more than half the total optimum, i.e., the flow value of a maximum flow over time with inflow rate $d_1 + d_2$. This can be achieved by first *mirroring* the competitor's strategy (copying the strategy but interchanging the roles of e_1 and e_2) and then shifting some flow in the beginning of the game. This shift changes the points in time r_i when the last particles arriving at t just in time, enter edge e_i for $i = 1, 2$. As the new values \hat{r}_1 and \hat{r}_2 are not equal anymore the responding player can modify the inflow rates between \hat{r}_1 and \hat{r}_2 in order to squeeze in a little more flow than the competitor, which will then be a little more than $\text{OPT} / 2$.

As in turn the competitor can again choose a strategy with a payoff of more than half the optimum, this immediately implies that a Nash equilibrium cannot exist. This idea for the network given in Figure 3a is visualized in Figures 3b and 3c.

Proof of Theorem 1. For the sake of simplicity we replace indexes e_i by i and p_j by j . E.g. $f_{1,2}$ denotes the inflow rate function of p_2 into e_1 . Furthermore, r_1 and r_2 denote the points in time when the last particles arriving at t before H enter e_1 and e_2 , respectively. We will use a hat to denote parameters that have altered with the change in strategy. For example, ρ will denote the initial and $\hat{\rho}$ the new payoff. The optimal value of a maximum flow is $\text{OPT} = \nu_1 \cdot (H - \tau_1) + \nu_2 \cdot (H - \tau_2)$. We show that to every strategy of player p_2 , player p_1 can choose a response strategy that yields a payoff of strictly more than half the optimum.

As in turn p_2 can again choose a strategy with a payoff of more than half the optimum, this immediately implies that a Nash equilibrium cannot exist. To achieve this, player p_1 mirrors p_2 's strategy. By this we mean that p_1 has the same strategy as p_2 , but the roles of the edges are interchanged. Figure 3b illustrates this behavior.

$$f_{1,1}^+(\theta) = f_{2,2}^+(\theta) \quad \text{and} \quad f_{2,1}^+(\theta) = f_{1,2}^+(\theta) \quad \text{for } \theta \in [0, H).$$

Both total inflow rates into the edges are equal and exceed the capacities, i.e., $f_{1,1}^+(\theta) + f_{1,2}^+(\theta) = f_{2,1}^+(\theta) + f_{2,2}^+(\theta) > \nu_1 = \nu_2$ for $\theta \in [0, H)$. It is easy to see that both payoffs are $\text{OPT}/2$. As capacities are exceeded, queues build up from the very beginning. At time $r_1 = r_2 = \text{OPT}/(d_1 + d_2)$ the last particles that will reach t in time enter the network. The response strategy of p_1 now consists of reallocating a little flow to benefit from deferred \hat{r}_1 and \hat{r}_2 : W.l.o.g. let p_2 send not more than half of her supply rate d_2 at any time in $[r_1 - \varepsilon, r_1)$ for $\varepsilon > 0$, into e_1 . That means player p_1 sends not less than half of d_1 into e_1 . (If no such ε exists, due to some crazy function behavior, one can consider averages instead.) Player p_1 reallocates some $\delta > 0$ flow units from e_1 to e_2 . By doing so, \hat{r}_2 is shifted to $r_2 - \frac{\delta}{\nu_2}$. Meanwhile we want that $\frac{\delta}{\nu_2} \leq \varepsilon$, that the δ flow units are taken before time $r_2 - \varepsilon$ and that the queue of e_1 is never empty for $\theta > 0$. Since there has to exist p_1 -flow of positive measure up to time r_1 on e_1 (in case the only flow p_1 sends into e_1 is within $[r_1 - \varepsilon, r_1)$ we can choose a smaller ε), we can clearly set δ small enough to fulfill these conditions. As a result, \hat{r}_1 is postponed, i.e., $\hat{r}_1 > \hat{r}_2$. Conclusively, every flow sent until time \hat{r}_2 reaches t before H . Hence, the payoffs of both players belonging to $[0, \hat{r}_2)$ are equal. After time \hat{r}_2 , player p_1 can pump all her flow into e_1 , since it is still eligible to reach t before H . See Figure 3c for an illustration. Therefore,

$$\hat{f}_{1,1}^+(\theta) := d_1 \begin{cases} > d_2/2 \geq f_{1,2}^+(\theta) & \theta \in [\hat{r}_2, r_1) \\ \geq f_{1,2}^+(\theta) & \theta \geq r_1. \end{cases}$$

Hence, p_1 's inflow rate into e_1 is strictly greater than p_2 's during a time period of positive measure, which shows that the payoff of p_1 is strictly larger than that of p_2 . As the sum of the payoffs equals OPT we have that $\hat{r}_1 > \text{OPT}/2$. ◀

4 Information on Exit Times

The absence of a Nash equilibrium for temporal information only was mainly due to the theoretical information advantage of the deviating player. Player p_1 can respond to p_2 's strategy, while p_2 does not see p_1 's moves and is unable to react. As it is very natural to require inter-player reactions over time, we extend the information by the current congestion of the edges in form of the exit times $T(\theta) := (T_e(\theta))_{e \in E}$. Formally, we define

$$I_j(\theta, f) := (\theta, T(\theta)) \in \mathcal{I} := [0, H) \times \mathbb{R}_{\geq 0}^E \quad \text{for all } j \in P.$$

The reason for choosing exit times over waiting times or queue sizes, which both contain the same information about the congestion in the networks as the exit times, is that the exit times are non-decreasing. As the exit time $T_e(\theta) = \theta + \tau_e + \frac{1}{\nu_e} \sum_{j \in P} (F_{e,j}^+(\theta) - F_{e,j}^-(\theta + \tau_e))$ depends directly on the cumulative flows the equations system (5) becomes a system of differential equations. In order to show existence and uniqueness we use the Picard-Lindelöf theorem. For this reason, we require the strategies to be locally Lipschitz-continuous from the right in order to ensure uniqueness. More formally, we say a strategy $g_{e,j}$ is *right-Lipschitz* if for every $I \in \mathcal{I}$ there exists an $L > 0$ such that there exists an $\varepsilon > 0$ with $|g_{e,j}(I) - g_{e,j}(I + x)| \leq L \cdot \|x\|$ for all $x \in [0, \varepsilon] \times [0, \varepsilon]^E$.

Existence and uniqueness. First we show, that in this setting every right-Lipschitz strategy profile yields a unique feasible flow over time.

► **Theorem 2.** *For all right-Lipschitz strategy profiles $g = (g_j)_{j \in P}$ of an atomic splittable flow over time game with information on exit times, there exists a unique feasible flow over time f^+ satisfying (5).*

Proof. We will construct the feasible flow over time satisfying (5) step by step starting with the empty flow over time $f^+ \equiv 0$ up to time 0. We define a *restricted flow over time* on the interval $[0, a)$ to be a vector of Lebesgue-integrable functions $(f_{e,j}^+)_{e \in E, j \in P}$, such that all flow conditions hold for all times in $[0, a)$.

Suppose we are given a unique restricted feasible flow over time satisfying (5) on the interval $[0, a)$ for some $a \in [0, H]$. If $a = H$, we are done. It is possible to determine the queue lengths $z(a) = (z_e(a))_{e \in E}$, the waiting times $q(a) = (q_e(a))_{e \in E}$ and the exit times $T(a) = (T_e(a))_{e \in E}$ based on $f^+|_{[0,a)}$. Hence, we can also evaluate $g(a, T(a))$. In order to further extend the flow over time, we specify an interval $[a, b) \subseteq [0, H]$. The right endpoint $b > a$ has to be small enough to ensure that $g_{e,j}(\theta, T(\theta))$ is Lipschitz-continuous for $\theta \in [a, b)$ and for all $e \in E$ and $j \in P$. We can find such b as all $g_{e,j}$ are right-Lipschitz, the exit time functions T_e are non-decreasing and continuous in θ .

We obtain the following initial value problem:

$$\begin{aligned} f_{e,j}^+(a) &= g_{e,j}(a, T(a)) \cdot C_u(a), \\ f_{e,j}^+(\theta) &= g_{e,j}(\theta, (\theta + \tau_{e'} + \frac{1}{\nu_{e'}} \sum_{j' \in P} F_{e',j'}^+(\theta) - F_{e',j'}^-(\theta + \tau_{e'}))_{e' \in E}) \cdot C_u(\theta), \end{aligned}$$

for all $j \in P$ and $e = uv \in E$. Here, $C_u(\theta) \geq 0$ is the total inflow rate into node u except for t where it is 0 (as stated in (5)). Since the transit times are strictly positive $C_u: [a, b) \rightarrow \mathbb{R}$ is completely determined by the restricted feasible flow over time on $[0, a)$ as long as $b - a < \tau_e$. In addition, C_u is right-Lipschitz, therefore, we can choose b small enough, such that C_u is Lipschitz-continuous on $[a, b)$. Furthermore, if $q_e(\theta) > 0$ we have that $F_{e',j'}^-(\theta + \tau_{e'})$ is also determined from the past as long as $b - a < q_e(\theta)$ or in the case of $q_e(\theta) = 0$, we have that $F_{e',j'}^-(\theta + \tau_{e'}) = \sum_{j' \in P} F_{e',j'}^+(\theta)$ as required by (2).

Since the exit times T depend Lipschitz-continuous on F^+ , also the right-side depends Lipschitz-continuous on F^+ . Hence, the Picard-Lindelöf theorem guarantees the existence of a unique solution. It is easy to see, that extending f by this solution yields a restricted feasible flow over time on $[0, b)$. Flow conservation is fulfilled as $\sum_{e \in \delta_u^+} g_{e,j}(\theta, T(\theta)) = 1$ and (2), (3) and (4) are satisfied as we implicitly define the outflow rates accordingly.

It remains to show that it is possible to repeat the process until we cover the whole interval $[0, H)$. Let $b_i, i = 1, 2, 3, \dots$, be the right endpoints during this extension process. The sequence might converge to $\lim_{i \rightarrow \infty} b_i = b_\infty < H$. Existence and uniqueness are thus provided on $[0, b_\infty)$. But this means, we can apply the extension process for $a = b_\infty$. As we can always continue this process from a limit point, we can apply this transfinite induction to obtain a unique feasible flow over time on $[0, H)$. ◀

This theorem shows that we obtain a well-defined atomic splittable flow over time game as long as we only consider strategies that are not too wild. It is worth noting that right-Lipschitz functions can have infinitely many jumps and that we cannot hope for much more general strategy-functions as argued in the following. Suppose we allow for strategies that are not continuous from the right. We end up with the following problem: Consider a game with only one player p_1 and a network consisting of two edges e_1, e_2 both from s_{p_1} to t_{p_1}

and with $\nu_{e_1} < d_{p_1}$. The strategy with $g_{e_1, p_1}(\theta, T(\theta)) = 1$ if $T_{e_1}(\theta) \leq \theta + \tau_{e_1}$ and 0 otherwise, means, that if there is no queue on e_1 the players sends all its flow into e_1 (which causes a queue to build up) but if there is a positive queue she sends nothing (which causes any positive queue to decrease). This strategy is not right-continuous in T as $g_{e_1, 1}$ switches from 1 to 0 as soon as $T > \tau_{e_1} + \theta$, and clearly, it cannot lead to a feasible flow over time.

Existence of Nash equilibria in parallel-edge networks. Unfortunately, the task to show the existence of a Nash equilibrium in this setting turns out to be quite challenging. For this reason we only show the existence of Nash equilibria for simple networks: For the remaining of this section we consider parallel-edge networks with two nodes $s = s_j$ and $t = t_j$, $j \in P$. We obtain

$$\Sigma_j = \left\{ g_j : \mathcal{I} \rightarrow [0, d_j]^E \mid \begin{array}{l} g_{e,j} \text{ is right-Lipschitz for all } e \in E \\ \text{and } \sum_{e \in E} g_{e,j}(I) = 1 \text{ for all } I \in \mathcal{I} \end{array} \right\}.$$

This leads to an ‘‘over time’’ version of *atomic splittable singleton games* as they were studied in by Harks and Timmermans [23]. As an additional motivation, it is worth noting, that these restricted networks, become more meaningful if, instead of a routing game, we consider a throughput-scheduling problem [42]. Suppose the edges represent machines on which competing players want to maximize their throughput. The jobs can run in parallel on multiple machines up to some maximum rate of d_j and each machine has a maximal service rate of ν_e and individual time horizons $H - \tau_e$. A very similar model without the game-theoretical aspect is for example studied by Armony and Bambos [2].

In order to obtain a Nash equilibrium, it is worth noting that players do not care on which edge their flow is sent, as long as it arrives at the destination before H . To model a suitable strategy, we introduce the set of *active edges* $E'(T) := \{ e \in E \mid T_e < H \}$ on which flow still arrives at t before H , depending on the exit times. The resulting strategy for a player $j \in P$ on edge $e \in E$ could look as follows:

$$g_{e,j}(\theta, T) := \begin{cases} \frac{\nu_e}{\sum_{e' \in E'(T)} \nu_{e'}} & \text{if } e \in E'(T), \\ 0 & \text{if } E'(T) \neq \emptyset \text{ and } e \notin E'(T), \\ \frac{\nu_e}{\sum_{e' \in E} \nu_{e'}} & \text{if } E'(T) = \emptyset. \end{cases} \quad (6)$$

The third case is not of importance for the player, as none of the flow sent into the network will arrive in time anymore. As $E'(T)$ stays constant for small increases of T , the same is true for $g_{e,j}$. Since $\sum_{e \in E} g_{e,j}(\theta, T) = 1$, we have $g_j \in \Sigma_j$.

We will show that this strategy profile leads to a Nash equilibrium. For this we first show in Lemma 3 that the given strategy profile leads to a total payoff equal to the system optimum OPT (the value of a maximum flow over time with inflow rate $\sum_{j \in P} d_j$). Afterwards, we determine in Lemma 4 that the payoff for each player given the strategy profile in (6) is a fixed share of the total payoff. Finally, we argue that none of the players has an incentive to deviate from the given strategy profile, since shares cannot be increased.

For the remaining of this section, let r_e be the point in time when the (first) particle that arrives at t at time H enters edge e . Formally, $r_e := \min T_e^{-1}(H)$. Hence, $T_e(\theta) < H$ for any $\theta < r_e$ and $T_e(\theta) \geq H$ for any $\theta \geq r_e$.

► **Lemma 3.** *For the strategy profile given in (6) the sum of the payoffs of all players equals the system optimum OPT.*

4:12 Atomic Splittable Flow Over Time Games

Proof. To prove this, we allow the network to have transit times that equal 0. We split the set of instances into three cases

Case 1: $\sum_{j \in P} d_j \geq \sum_{e \in E} \nu_e$.

W.l.o.g. we assume $\tau_e < H$; otherwise the edge would be superfluous and could be deleted.

We have $\text{OPT} = \sum_{e \in E} \nu_e \cdot (H - \tau_e)$ as shown in [43]. Since all players route their flow proportionally to the capacities of the active edges, we have

$$\sum_{j \in P} f_{e,j}^+(\theta) = \frac{\nu_e}{\sum_{e' \in E'(T(\theta))} \nu_{e'}} \cdot \sum_{j \in P} d_j \geq \nu_e \quad \text{for all } \theta \text{ with } T_e(\theta) < H.$$

With this at hand, we can state the total outflow during the whole game:

$$\begin{aligned} \sum_{j \in P} \rho_j &= \sum_{e \in E} \sum_{j \in P} F_{e,j}^-(H) = \sum_{e \in E} \int_0^H \sum_{j \in P} f_{e,j}^-(\theta) d\theta \\ &= \sum_{e \in E} \left(\int_0^{\tau_e} 0 d\theta + \int_{\tau_e}^H \nu_e d\theta \right) = \sum_{e \in E} (H - \tau_e) \cdot \nu_e = \text{OPT}. \end{aligned}$$

Case 2: $\sum_{j \in P} d_j < \sum_{e \in E} \nu_e$ and $\tau_e = 0$ for all $e \in E$.

We can easily see that $\text{OPT} = H \cdot \sum_{j \in P} d_j$ and

$$\sum_{j \in J} f_{e,j}^+(\theta) = \frac{\nu_e}{\sum_{e' \in E} \nu_{e'}} \cdot \sum_{j \in P} d_j < \nu_e \quad \text{for all } \theta < H.$$

Since no queues build up, all edges stay active for all $\theta \in [0, H)$. Therefore, it holds that

$$\sum_{j \in P} \rho_j = \sum_{e \in E} \int_0^H \sum_{j \in P} f_{e,j}^-(\theta) d\theta = \sum_{e \in E} \sum_{j \in P} H \cdot d_j \cdot \frac{\nu_e}{\sum_{e' \in E} \nu_{e'}} = H \cdot \sum_{j \in P} d_j = \text{OPT}.$$

Case 3: $\sum_{j \in P} d_j < \sum_{e \in E} \nu_e$ and there exists an $e \in E$ with $\tau_e > 0$.

We assume that $\tau_e < H$ for all $e \in E$. Let e^* be the first edge to drop out of $E'(T(\theta))$, i.e., r_{e^*} is minimal among all r_e . As in the first phase no queues build up (see Case 2), we have $r_{e^*} = H - \tau_{e^*}$, which means that τ_{e^*} is maximal among all τ_e . Emphasis should be put on the fact that the whole flow sent into the network up to time r_{e^*} arrives at t in time; a volume of $\sum_{j \in P} d_j \cdot (H - \tau_{e^*})$ in total. Hence, the system optimum cannot perform any better until r_{e^*} . To show that after time r_{e^*} , the summed payoffs correspond to the system optimum as well, we reduce the remaining instance. First, we remove e^* from the set of edges, i.e., $\hat{E} = E \setminus \{e^*\}$. Second, we shift the time axis r_{e^*} time units back. By that we mean that the new time 0 corresponds to r_{e^*} in the former instance and $\hat{H} = H - r_{e^*} = \tau_{e^*}$. Everything else remains untouched, in particular all queues are empty at r_{e^*} . This instance is strictly smaller, which means that eventually the reduction process must end because either we obtain $\sum_{j \in P} d_j \geq \sum_{e \in \hat{E}} \nu_e$ (Case 1), $\tau_e = 0$ for all $e \in \hat{E}$ (Case 2) or we reach $|\hat{E}| = 1$ in which case the total payoff trivially equals OPT. ◀

► **Lemma 4.** For the strategy profile given in (6), the payoff of player j is given by

$$\rho_j = \text{OPT} \cdot \frac{d_j}{\sum_{j' \in P} d_{j'}}.$$

Proof. For $\theta \in [0, r_e)$ we have $\sum_{j \in P} f_{e,j}^+(\theta) = \sum_{j \in P} d_j \cdot g_{e,j}(\theta, T(\theta)) > 0$. We want to examine the outflow rates for $\phi \in [\tau_e, H)$. With (4) and $\theta = \max T_e^{-1}(\phi) < r_e$ we obtain for all $e \in E$ and $j \in P$ that

$$\begin{aligned} f_{e,j}^-(\phi) &= \frac{f_{e,j}^+(\theta)}{\sum_{j' \in P} f_{e,j'}^+(\theta)} \cdot \sum_{j' \in P} f_{e,j'}^-(\phi) \\ &= \frac{d_j \cdot g_{e,j}(\theta, T(\theta))}{\sum_{j' \in P} d_{j'} \cdot g_{e,j'}(\theta, T(\theta))} \cdot \sum_{j' \in P} f_{e,j'}^-(\phi) = \frac{d_j}{\sum_{j' \in P} d_{j'}} \cdot \sum_{j' \in P} f_{e,j'}^-(\phi). \end{aligned}$$

Taking the integral over $[0, H]$, summing over all edges $e \in E$ and using Lemma 3, yields

$$\rho_j = \sum_{e \in E} F_{e,j}^-(H) = \frac{d_j}{\sum_{j' \in P} d_{j'}} \cdot \sum_{e \in E} \sum_{j' \in P} F_{e,j'}^-(\phi) = \frac{d_j}{\sum_{j' \in P} d_{j'}} \cdot \text{OPT}. \quad \blacktriangleleft$$

With these two lemmas we can finally prove the following theorem.

► **Theorem 5.** *The strategy profile $(g_j)_{j \in P}$ given by (6) is a Nash equilibrium.*

Proof. We want to observe what happens if one player j^* deviates from her strategy in an ex ante manner. For this let $r := \max_{e \in E} r_e$ be the point in time when the last edge becomes inactive for the modified strategy-profile and let $\rho := \sum_{j \in P} \rho_j$ be the total amount of flow arriving in time. Clearly, $\rho \leq \text{OPT}$.

During $[0, r]$ every non-deviating player $j \in P \setminus \{j^*\}$ sends all her flow into edges that are still active, and therefore, all this flow arrives in time. Hence, her payoff is given by $\rho_j = d_j \cdot r$.

Since no flow that enters after r can possibly arrive in time, player j^* can achieve at most a payoff of $d_{j^*} \cdot r$. It follows that her share of the total amount of flow arriving in time is upper bounded by $\frac{d_{j^*}}{\sum_{j \in P} d_j}$, i.e.,

$$\rho_{j^*} \leq \rho \cdot \frac{d_{j^*}}{\sum_{j \in P} d_j} \leq \text{OPT} \cdot \frac{d_{j^*}}{\sum_{j \in P} d_j}.$$

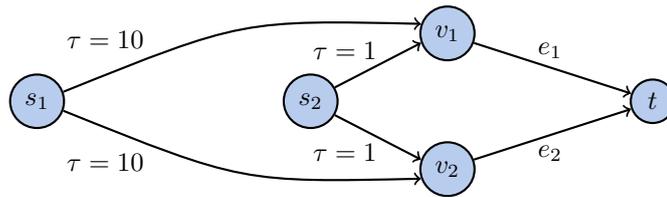
By Lemma 4 the right-side equals the payoff of player j^* when choosing g_{j^*} as strategy. Hence, she cannot improve her payoff by deviating, which shows that $(g_j)_{j \in P}$ is indeed an equilibrium. \blacktriangleleft

We observe the following: As long as the players constantly choose from the set of active edges and the summed payoff equals OPT, i.e., no queue runs dry in case of a restricted network, the players' payoffs stay the same. Hence, there is a whole class of Nash equilibria.

Next we want to show that this characterizes all possible Nash equilibria, which implies that the price of anarchy is 1.

► **Theorem 6.** *For atomic splittable flow over time games on parallel-edge networks where exit times are provided as information with right-Lipschitz strategies, the price of anarchy is 1.*

Proof. The key observation is that every player j 's share of the total payoff $\sum_{j' \in P} \rho_{j'}$ is at least $d_j / \sum_{j' \in P} d_{j'}$ as long as the player only sends flow into active edges $E'(T(\theta))$. Also, the total payoff is never decreased when a player shifts inflow from an inactive edge to an active edge, since flow sent into inactive edges does not arrive in time. Note here that due to



■ **Figure 4** A network with two non-symmetric players. We have $t = t_1 = t_2$. Player 1 must commit to a split of her inflow rate at time θ before knowing the relevant information on the congestion on e_1 and e_2 at time $\theta + 10$. Player 2 might use this to her advantage, which could lead to the non-existence of Nash equilibria.

the flow dynamics, the cumulative outflow function of an edge depends non-decreasingly on the cumulative inflow function of the edge (more precisely, $F_{j,e}^-(\theta) \geq \hat{F}_{j,e}^-(\theta)$ for all $\theta \in [0, H]$ if $F_{j,e}^+(\theta) \geq \hat{F}_{j,e}^+(\theta)$ for all $\theta \in [0, H]$). Suppose we have a strategy profile g such that the total payoff $\sum_{j \in P} \rho_j$ is strictly smaller than OPT. Either one of the players' shares of the total payoff is strictly less than $d_j / \sum_{j' \in P} d_{j'}$, so this player can improve, or all players have a share of $d_j / \sum_{j' \in P} d_{j'}$. But then there has to be an edge e where flow is wasted, which means that the queue on e is empty and the total inflow rate is strictly smaller than the capacity for a time span of positive measure when e is still active; instead, flow is sent into inactive edges or edges with a queue, which hinders later flow to get to t in time. Hence, j can improve by shifting flow from an inactive edge or edge with a queue, as this increases the total payoff but does not decrease her share. Hence, in both cases g was not a Nash equilibrium, which shows that the price of anarchy is 1. ◀

5 Further Research

The topic opens up a multitude of further research directions. First of all, either proving or disproving the existence of Nash equilibria for more general networks for games with information on the exit times. As the exit times do not cover all information that might be necessary for the players to react, a responding player might have an advantage. This is especially critical in games with non-symmetric players. To illustrate this difficulty consider the network in Figure 4.

For this reason an interesting research direction would be to identify more general network classes for which a Nash equilibrium still exists. Symmetric games where all players share the same origin and the same destination might be a necessary restriction.

Additionally, the dependencies among different objective functions have not yet been understood very well: Shedding light on whether the results for the maximum flow over time objective translate to a quickest flow objective (maybe via binary-search) would be very interesting. It might even be possible to consider the average arrived flow or the average arrival time as payoff functions, which could then be compared to an earliest arrival flow.

Finally, cooperative and non-cooperative models might be mixed in order to assess how coalitions and selfish particles behave together (as e.g. in [7] for the static case).

References

- 1 Eitan Altman, Tamer Bacsar, Tania Jimenez, and Nahum Shimkin. Competitive routing in networks with polynomial costs. *IEEE Transactions on Automatic Control*, 47(1):92–96, 2002.
- 2 Mor Armony and Nicholas Bambos. Queueing dynamics and maximal throughput scheduling in switched processing systems. *Queueing systems*, 44(3):209–252, 2003.

- 3 U. Bhaskar, L. Fleischer, and E. Anshelevich. A stackelberg strategy for routing flow over time. *Games and Economic Behavior*, 92:232–247, 2015.
- 4 Umang Bhaskar, Lisa Fleischer, Darrell Hoy, and Chien-Chung Huang. On the uniqueness of equilibrium in atomic splittable routing games. *Mathematics of Operations Research*, 40(3):634–654, 2015.
- 5 Umang Bhaskar and Phani Raj Lolakapuri. Equilibrium computation in atomic splittable routing games. In *26th Annual European Symposium on Algorithms (ESA 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.
- 6 R. E. Burkard, K. Dlaska, and B. Klinz. The quickest flow problem. *Zeitschrift für Operations Research*, 37(1):31–58, 1993.
- 7 Stefano Catoni and Stefano Pallottino. Traffic equilibrium paradoxes. *Transportation Science*, 25(3):240–244, 1991.
- 8 R. Cominetti, J. Correa, and O. Larré. Existence and uniqueness of equilibria for flows over time. In *International Colloquium on Automata, Languages, and Programming*, pages 552–563. Springer, 2011.
- 9 R. Cominetti, J. Correa, and N. Olver. Long-term behavior of dynamic equilibria in fluid queuing networks. *Operations Research*, 2021.
- 10 Roberto Cominetti, José R. Correa, and Omar Larré. Dynamic equilibria in fluid queueing networks. *Operations Research*, 63(1):21–34, 2015.
- 11 Roberto Cominetti, José R. Correa, and Nicolás E. Stier-Moses. The impact of oligopolistic competition in networks. *Operations Research*, 57(6):1421–1437, 2009.
- 12 José Correa, Andrés Cristi, and Tim Oosterwijk. On the price of anarchy for flows over time. In *Proceedings of the 2019 ACM Conference on Economics and Computation, EC '19*, pages 559–577, New York, 2019. Association for Computing Machinery.
- 13 José Correa and Nicolás E. Stier-Moses. Wardrop equilibria. *Wiley encyclopedia of operations research and management science*, 2010.
- 14 Lisa Fleischer and Éva Tardos. Efficient continuous-time dynamic network flow algorithms. *Operations Research Letters*, 23(3):71–80, 1998.
- 15 L. R. Ford and D. R. Fulkerson. Constructing maximal dynamic flows from static flows. *Operations research*, 6:419–433, 1958.
- 16 L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.
- 17 David Gale. Transient flows in networks. *Michigan Mathematical Journal*, 6(1):59–63, 1959.
- 18 Lukas Graf and Tobias Harks. The price of anarchy for instantaneous dynamic equilibria. In *International Conference on Web and Internet Economics*, pages 237–251. Springer, 2020.
- 19 Lukas Graf and Tobias Harks. A finite time combinatorial algorithm for instantaneous dynamic equilibrium flows. *Mathematical Programming*, pages 1–32, 2022.
- 20 Lukas Graf, Tobias Harks, and Leon Sering. Dynamic flows with adaptive route choice. *Mathematical Programming*, 2020.
- 21 Tobias Harks. Stackelberg strategies and collusion in network games with splittable flow. *Theory of Computing Systems*, 48(4):781–802, 2011.
- 22 Tobias Harks, Britta Peis, Daniel Schmand, Bjoern Tauer, and Laura Vargas Koch. Competitive packet routing with priority lists. *ACM Transactions on Economics and Computation (TEAC)*, 6(1):4, 2018.
- 23 Tobias Harks and Veerle Timmermans. Equilibrium computation in atomic splittable singleton congestion games. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 442–454. Springer, 2017.
- 24 Tobias Harks and Veerle Timmermans. Uniqueness of equilibria in atomic splittable polymatroid congestion games. *Journal of Combinatorial Optimization*, 36(3):812–830, 2018.
- 25 Alain Haurie and Patrice Marcotte. On the relationship between nash-cournot and wardrop equilibria. *Networks*, 15(3):295–308, 1985.

- 26 Martin Hoefer, Vahab S Mirrokni, Heiko Röglin, and Shang-Hua Teng. Competitive routing over time. In *International Workshop on Internet and Network Economics*, pages 18–29. Springer, 2009.
- 27 J. Israel and L. Sering. The impact of spillback on the price of anarchy for flows over time. In *International Symposium on Algorithmic Game Theory*, pages 114–129. Springer, 2020.
- 28 Max Klimm and Philipp Warode. Complexity and parametric computation of equilibria in atomic splittable congestion games via weighted block laplacians. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2728–2747. SIAM, 2020.
- 29 Ronald Koch and Martin Skutella. Nash equilibria and the price of anarchy for flows over time. *Theory of Computing Systems*, 49(1):71–97, 2011.
- 30 Patrice Marcotte. Algorithms for the network oligopoly problem. *Journal of the Operational Research Society*, 38(11):1051–1065, 1987.
- 31 E. Minieka. Maximal, lexicographic, and dynamic network flows. *Operations Research*, 21(2):517–527, 1973.
- 32 N. Olver, L. Sering, and L. Vargas Koch. Continuity, uniqueness and long-term behavior of Nash flows over time. In *2021 IEEE 62st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2021.
- 33 Britta Peis, Bjoern Tauer, Veerle Timmermans, and Laura Vargas Koch. Oligopolistic competitive packet routing. In *18th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.
- 34 H. Pham and L. Sering. Dynamic equilibria in time-varying networks. In *International Symposium on Algorithmic Game Theory*, pages 130–145. Springer, 2020.
- 35 J Ben Rosen. Existence and uniqueness of equilibrium points for concave n-person games. *Econometrica: Journal of the Econometric Society*, pages 520–534, 1965.
- 36 Tim Roughgarden. Selfish routing with atomic players. *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '05)*, pages 1184–1185, 2005.
- 37 Tim Roughgarden and Florian Schoppmann. Local smoothness and the price of anarchy in splittable congestion games. *Journal of Economic Theory*, 156:317–342, 2015.
- 38 Tim Roughgarden and Éva Tardos. How bad is selfish routing? *Journal of the ACM*, 49(2):236–259, 2002.
- 39 L. Sering. *Nash flows over time*. Technische Universitaet Berlin (Germany), 2020.
- 40 L. Sering and M. Skutella. Multi-source multi-sink Nash flows over time. In *18th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*, volume 65, pages 12:1–12:20, 2018.
- 41 L. Sering and L. Vargas Koch. Nash flows over time with spillback. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 935–945. SIAM, 2019.
- 42 Jiří Sgall. Open problems in throughput scheduling. In *European Symposium on Algorithms*, pages 2–11. Springer, 2012.
- 43 Martin Skutella. An introduction to network flows over time. In *Research trends in combinatorial optimization*, pages 451–482. Springer, 2009.
- 44 W. S. Vickrey. Congestion theory and transport investment. *The American Economic Review*, 59(2):251–260, 1969. URL: <http://www.jstor.org/stable/1823678>.
- 45 John Glen Wardrop. Some theoretical aspects of road traffic research. *Proceedings of the Institution of Civil Engineers*, 1(3):325–362, 1952.

Faster Exploration of Some Temporal Graphs

Duncan Adamson ✉

Department of Computer Science, Reykjavik University, Iceland

Vladimir V. Gusev ✉

Materials Innovation Factory, University of Liverpool, UK

Department of Computer Science, University of Liverpool

Dmitriy Malyshev ✉

Laboratory of Algorithms and Technologies for Network Analysis, HSE University,

Nizhny Novgorod, Russian Federation

Viktor Zamaraev ✉

Department of Computer Science, University of Liverpool, UK

Abstract

A temporal graph $G = (G_1, G_2, \dots, G_T)$ is a graph represented by a sequence of T graphs over a common set of vertices, such that at the i^{th} time step only the edge set E_i is active. The temporal graph exploration problem asks for a shortest temporal walk on some temporal graph visiting every vertex. We show that temporal graphs with n vertices can be explored in $O(kn^{1.5} \log n)$ days if the underlying graph has treewidth k and in $O(n^{1.75} \log n)$ days if the underlying graph is planar. Furthermore, we show that any temporal graph whose underlying graph is a cycle with k chords can be explored in at most $6kn$ days. Finally, we demonstrate that there are temporal realisations of sub cubic planar graphs that cannot be explored faster than in $\Omega(n \log n)$ days. All these improve best known results in the literature.

2012 ACM Subject Classification Mathematics of computing → Paths and connectivity problems

Keywords and phrases Temporal Graphs, Graph Exploration

Digital Object Identifier 10.4230/LIPIcs.SAND.2022.5

Funding *Duncan Adamson*: Funded by the Leverhulme trust.

Vladimir V. Gusev: Funded by the Leverhulme trust.

Dmitriy Malyshev: The work of Dmitriy Malyshev was conducted within the framework of the Basic Research Program at the National Research University Higher School of Economics (HSE).

1 Introduction

In many real world settings, networks are not static objects but instead have unstable connections that vary with time. Temporal graphs provide a model for such time-varying networks. Formally, a *temporal graph* \mathcal{G} is a sequence $(G_1, G_2, G_3, \dots, G_T)$ of undirected graphs, called *snapshots*, that all share the same vertex set V , but whose edge sets $E_1, E_2, E_3, \dots, E_T$, respectively, may differ. The number $T + 1$ is called the *lifetime* of \mathcal{G} and we refer to i , $0 \leq i \leq T$, as a *time* i or *day* i . The graph $G = (V, E_1 \cup E_2 \cup \dots \cup E_T)$ is called the *underlying graph* of \mathcal{G} , and \mathcal{G} is said to be a *temporal realisation* of G . A pair (e, i) , where $e \in E_i$ is called a *time edge* of \mathcal{G} . A *temporal walk* from $v_1 \in V$ starting at time t to $v_k \in V$ is an alternating sequence of vertices and time edges $v_1, (e_1, i_1), v_2, \dots, v_{k-1}, (e_{k-1}, i_{k-1}), v_k$ such that $e_j = \{v_j, v_{j+1}\} \in E_{i_j}$ for $0 \leq j \leq k - 1$ and $t \leq i_1 < i_2 < \dots < i_{k-1}$. The time $i_{k-1} + 1$ is called the *arrival time* of the walk.

Motivated by the central role of the TRAVELLING SALESMAN problem in the world of static graphs, Michail and Spirakis [7] introduced and initiated the study of the natural temporal analogue called the TEMPORAL GRAPH EXPLORATION problem (TEXP for brevity). The goal of TEXP is to compute a temporal walk with the *earliest* arrival time that starts in a given vertex $s \in V$ and visits (i.e., *explores*) all vertices of the temporal graph. It is often



© Duncan Adamson, Vladimir V. Gusev, Dmitriy Malyshev, and Viktor Zamaraev; licensed under Creative Commons License CC-BY 4.0

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).

Editors: James Aspnes and Othon Michail; Article No. 5; pp. 5:1–5:10

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

convenient to describe a construction of an exploration temporal walk as the actions of an agent that is initially located at some starting vertex s and that can in every day i either stay at its current node or move to a node u that is adjacent to its current node in E_i . In the latter case, the agent *departs* from the current vertex at time i and *arrives* at u at time $i + 1$.

The decision version of TEXP in which one has to decide if at least one exploration schedule exists in a given temporal graph from a given starting vertex is an **NP**-complete problem [7]. In fact, this decision problem remains **NP**-complete even if the underlying graph has pathwidth 2 and every snapshot is a tree [2], or even if the underlying graph is a star and the exploration has to start and end at the center of the star [1].

Michail and Spirakis [7] proved that TEXP admits no $(2 - \varepsilon)$ -approximation algorithm for any $\varepsilon > 0$, unless **P=NP**. In other words, there is no polynomial time algorithm that outputs an exploration schedule whose arrival time is at most $(2 - \varepsilon)$ times the arrival time of an optimal exploration schedule. This was substantially strengthened by Erlebach et. al. [3] who established **NP**-hardness of $n^{1-\varepsilon}$ -approximation for any $\varepsilon > 0$. In fact, the result was shown for *always-connected* temporal graphs, i.e., temporal graphs in which every snapshot is a connected graph. This connectedness assumption makes the above inapproximability result tight, because on the one hand, obviously, any exploration cannot be done faster than in $n - 1$ days, and on the other hand any always-connected temporal graph can be explored in at most n^2 days [7].

The strong inapproximability result for TEXP on always-connected temporal graphs motivated the study into bounds on the length of fastest exploration schedules for such temporal graphs and the present work contributes to this line of research. For convenience, from now on, unless specified otherwise we assume that every temporal graph is always-connected and has lifetime at least n^2 . In [3] Erlebach et. al. demonstrated that for some temporal graphs any exploration requires $\Omega(n^2)$ days, and thus showed that the upper bound of Michail and Spirakis [7] is asymptotically best possible. This result naturally led the investigation to consider restricted temporal graphs.

One natural way to restrict a temporal graph is to restrict its underlying graph. In this direction, Erlebach et. al. [3] showed that temporal realisations of planar graphs can be explored in $O(n^{1.8} \log n)$ days; temporal realisations of graphs of treewidth at most k can be explored in $O(k^{1.5} n^{1.5} \log n)$ days; temporal realisations of $2 \times n$ grids can be explored in $O(n \log^3 n)$ days. They also showed that temporal realisations of a cycle or a cycle with a single chord can be explored in $O(n)$ days, and conjectured that any temporal graph whose underlying graph is a cycle with at most k chords can be explored in no more than $f(k) \cdot n$ days, where $f(k)$ is some function. This conjecture was recently proved by Alamouti [8] with a factorial-type function $f(k) = k^2 k! e^k$. In [4] Erlebach et. al. proved that any temporal graph in which every snapshot is a bounded-degree graph (in particular, temporal realisations of bounded-degree graphs) can be explored in $O(n^{1.75})$ days. On the negative side, Erlebach et. al. [3] constructed temporal realisations of planar graphs of degree at most 4 that cannot be explored faster than in $\Omega(n \log n)$ days.

In [5] Erlebach and Spooner considered TEXP under another natural restriction on the input temporal graphs. Namely, they studied TEXP on *k-edge-deficient* temporal graphs, i.e., temporal graphs in which every snapshot is obtained from the underlying graph by removing at most k edges. They showed that *k-edge-deficient* and *1-edge-deficient* temporal graphs can be explored in $O(kn \log n)$ and $O(n)$ days, respectively, and constructed *k-edge-deficient* temporal graphs that cannot be explored faster than in $\Omega(n \log k)$ days.

■ **Table 1** Summary of the new algorithms we provide for exploring temporal graphs versus the previous best known bounds.

Setting	Known Bounds	Our Bounds
Cycles with k -Chords	$O(6k^2 \cdot k! \cdot (2e)^k n)$ ([8])	$O(kn)$
Treewidth- k graphs	$O(k^{1.5} n^{1.5} \log(n))$ ([3])	$O(kn^{1.5} \log(n))$
Planar graphs	$O(n^{1.8} \log(n))$ ([3])	$O(n^{1.75} \log(n))$

Our contribution

In this work we improve a number of bounds on exploration of temporal graphs with underlying graphs from restricted classes of graphs. Table 1 provides a summary of these results and how they compare to the best known current bounds. First, in Section 2 we show that a temporal realisation of a cycle with k chords can be explored in at most $6kn$ days. Next, in Section 3 we first strengthen the exploration bound of Erlebach et. al. [3] for temporal realisations of graphs admitting a (r, b) -division; then using this bound we prove that any temporal realisation of a planar graph can be explored in $O(n^{1.75} \log n)$ days and any temporal realisation of a graph of treewidth at most k can be explored in $O(kn^{1.5} \log n)$ days. Finally, in Section 4 we demonstrate that there are temporal realisations of planar graphs of degree at most 3 that cannot be explored faster than $\Omega(n \log n)$. The latter result is tight in the sense that temporal realisations of graphs of degree at most 2 are explorable in $O(n)$ days.

Notation and tools

For a vertex set $W \subseteq V$ we denote by $\mathcal{G}[W]$ the *temporal subgraph of \mathcal{G} induced by W* , i.e., the temporal graph $(G_1[W], G_2[W], G_3[W], \dots, G_T[W])$, where for a static graph G the notation $G[W]$ means the subgraph of G induced by W . In the our proofs we will employ two useful lemmas from [3].

► **Lemma 1** (reachability, [3]). *Let \mathcal{G} be a (not necessarily always-connected) temporal graph with a vertex set V . Let $U \subseteq V$ be a set of vertices of \mathcal{G} of size k , and let $u, v \in U$. If there exists a set of $k - 1$ snapshots each of which has a path from u to v that contains only vertices from U , then an agent can reach v starting from u and moving only in these $k - 1$ snapshots.*

Notice that a straightforward consequence of the above lemma is that a temporal graph can always be explored in at most n^2 days by visiting vertices in an arbitrary order and spending at most n days to move from one vertex to the next.

The following lemma is a general reduction that transforms a multi-agent exploration schedule to a single-agent one. In the multi-agent setting, there are several agents that all start at the same vertex and move or stay put in every day independently from each other. Similarly to the single-agent setting, the goal is to visit (explore) every vertex of the temporal graph by at least one agent as soon as possible.

► **Lemma 2** (multi-agent to single-agent, [3]). *Let G be a graph on n vertices. If any temporal realisation of G can be explored in t days with k agents, then any temporal realisation of G can be explored in $O((t + n)k \log n)$ days with one agent.*

2 Cycles with bounded number of chords

Erlebach et al. proved in [3] that a temporal realisation of a cycle can be explored in at most $2n - 2$ days, and this is a tight bound in the sense that there exist temporal realisations of cycles on n vertices for which any optimal exploration requires at least $2n - 3$ days. In the same work it was further shown that any temporal realisation of a cycle with one chord can be explored in at most $7n$ days. Furthermore, the authors conjectured that temporal realisations of cycles with a constant number of chords are explorable in $O(n)$ days. This conjecture was confirmed in [8] where the author has shown that a temporal realisation of a cycle with k chords can be explored in at most $6k^2 \cdot k! \cdot (2e)^k n$ days. In the present section we strengthen this result by showing that any temporal realisation of a cycle with k chords can be explored in $6kn$ days. We start with the following auxiliary lemma.

► **Lemma 3.** *Let $\mathcal{G} = (G_1, G_2, \dots, G_{2n})$ be an n -vertex temporal graph of lifetime $T = 2n$, and let G be the underlying graph of \mathcal{G} . Let $P = (v_1, v_2, \dots, v_\rho)$, $\rho \geq 1$, be a path in G such that every vertex of P , except possibly its endpoints v_1 and v_ρ , has degree 2 in G . Moreover, in every snapshot of \mathcal{G} at most one edge of P is absent. Then there exists a vertex $v \in V(G)$ such that all vertices of P can be explored starting from v .*

Proof. If there exists n snapshots in \mathcal{G} , in which all edges of P are present, then clearly the vertices of P can be explored in this n snapshots starting from any of the endpoints of P . It can therefore be assumed that there are less than n such snapshots, i.e., there are at least $n + 1$ snapshots in which exactly one edge of P is absent. We can therefore assume without loss of generality that the first $n + 1$ snapshots G_1, G_2, \dots, G_{n+1} of \mathcal{G} miss exactly one edge.

Observe that as every snapshot is connected and exactly one edge of P is absent in every snapshot, for every $i \in \{1, 2, \dots, n + 1\}$ the graph $H_i = G_i - \{v_1, \dots, v_{\rho-1}\}$ is connected. Therefore, by Lemma 1, in the temporal graph $\mathcal{H} = (H_1, H_2, \dots, H_{n+1})$ every vertex can reach any other vertex in at most $n - (\rho - 1) - 1 = n - \rho$ days. For every $i \in \{1, 2, \dots, \rho\}$, let J_i denote some fixed temporal walk from v_1 to v_ρ in \mathcal{H} that starts at time i and has the earliest arrival time. Note that the arrival time of J_i is at most $n - \rho + i$.

Now assume there are $n + 1$ agents a_1, a_2, \dots, a_{n+1} that are initially placed at the vertices of \mathcal{G} as follows: agent a_i is located at v_i for every $i \in \{0, 1, \dots, \rho - 1\}$ and all the other $n - \rho + 1$ agents $a_\rho, a_{\rho+1}, a_{\rho+2}, \dots, a_n$ are located at vertex v_ρ . Every day each agent will either move or stay at its current vertex. To describe the movement rules, let the *score* $\mu_t(a)$ of an agent a at time t as equal to the number vertices of P that a visited by time t . In particular, $\mu_2(a_i) = 1$ for every $i \in \{1, 2, \dots, n + 1\}$. Now, if the score $\mu_t(a)$ of an agent a at time t is $\rho + 1$, then a does not move. Otherwise the movement of the agent a at day t is determined according to the following rules:

1. *a is at vertex v_ρ at time t .* If $\mu_t(a)$ is the minimum among all agents that are currently at v_ρ , then a moves to $v_{\rho-1}$. Otherwise a stays at v_ρ . If there are multiple agents with the minimum value of $\mu_t(a)$, then only the agent with the minimum index moves and all other stay at v_ρ . If there is no edge between v_ρ and $v_{\rho-1}$ at time t , then the moving agent dies;
2. *a is at vertex v_i at time t , for some $i \in \{1, 2, 3, \dots, \rho - 1\}$.* Then a moves to v_{i-1} . As before, if there is no edge between v_i and v_{i-1} at time t , then a dies;
3. *a is at vertex v_1 at time t .* Then starting from time t the agent a moves according to the temporal walk J_t .

Observe that at every day at most one agent can die, and therefore after n days at least one of the agents survives. This leaves the problem of showing that any such agent has visited all vertices of P . To this end, let a_i be an agent that is alive after n days.

If $i \geq \rho$, then, according to the initial positions and the moving rules, a_i will start moving from vertex v_ρ at day $i - \rho$, and will visit one new vertex of P every day. Since $i \leq n$, after $i - \rho + \rho \leq n$ days a_i visits all vertices of P and stops moving.

Suppose now that $i < \rho$. According to the rules, after the first i days, the agent a_i moves along the path P and visits the vertices v_i, v_{i-1}, \dots, v_1 . After visiting all these vertices the score $\mu_i(a_i)$ of a_i at time i is equal to $i + 1$ and the agent continues to move following the temporal walk J_i . Let t^* be the time when a_i arrives at v_ρ . Observe that $\mu_{t^*}(a_i) = i + 2$ and $t^* \leq n - \rho + i$. By the end of day t^* , there could be at most $n - \rho - t^* + i$ agents at vertex v_ρ with a smaller score than the score $\mu_{t^*}(a_i)$ of a_i : at most $n - \rho + 1 - (t^* + 1) = n - \rho - t^*$ agents that were initially located at v_ρ and have not departed until the end of day t^* and at most i agents a_i, a_{i-1}, \dots, a_1 that arrived at v_ρ earlier or at the same time as a_i (and have not departed until the end of day t^*). All other agents at v_ρ at time t^* , if any, have larger scores. Hence, the agent a_i will depart from v_ρ no later than on day

$$t^* + (n - \rho - t^* + i) + 1 = n - \rho + i + 1 = n - (\rho - i - 1),$$

and therefore it will survive for further $\rho - i - 1$ days thus visiting the remaining $\rho - i - 1$ vertices $v_{\rho-1}, v_{\rho-2}, \dots, v_{i+1}$ of P . ◀

► **Theorem 4.** *A temporal realisation of a cycle with n vertices and k chords can be explored in at most $6kn$ days.*

Proof. Let \mathcal{G} be a temporal realisation of an n -vertex cycle with k chords and let G be the underlying graph of \mathcal{G} . Let us denote by C the underlying cycle of \mathcal{G} , and let a_1, a_2, \dots, a_s , $s \leq 2k$, be the distinct vertices of C , ordered according to their clockwise appearance on the cycle, which are incident with at least one of the chords. For every $i \in \{1, 2, \dots, s\}$, let P_i be the subpath of C that one obtains by following the cycle clockwise starting at a_i and ending at a_{i+1} , where the summation is modulo s .

Let $i \in \{1, 2, \dots, s\}$ be an arbitrary fixed index. Note that all internal vertices of P_i have degree 2 in the underlying graph G . This together with the connectivity of the snapshots of \mathcal{G} imply that at every day at most one edge of P_i is absent. Therefore, P_i and the temporal graph obtained by restricting \mathcal{G} to any sequence of $2n$ consecutive snapshots satisfy the assumptions of Lemma 3. Hence, the vertices of P_i can be visited during any sequence of $3n - 1$ consecutive snapshots: Using Lemma 3, in the first $n - 1$ snapshots we reach a vertex v guaranteed by Lemma 3, and in the subsequent $2n$ snapshots, by Lemma 3, we visit all the vertices of P_i starting from v . Since the index i was chosen arbitrarily, the above procedure can be repeated for each of the s paths, which implies that \mathcal{G} be explored in at most $2k(3n - 1) < 6kn$ days. ◀

3 Underlying graphs with (r, b) -divisions

In [3] Erlebach et al. showed that any temporal realisation of an n -vertex graph of treewidth k can be explored in $O(k^{1.5}n^{1.5} \log n)$ days, and any temporal realisation of an n -vertex planar graph can be explored in $O(n^{1.8} \log n)$ days. The key ingredient in the proofs of both results was the following

► **Theorem 5** (Theorem 4.3, [3]). *A temporal graph \mathcal{G} , whose underlying graph has a (r, b) -division¹, can be explored in $O((n + r^2b)^{\frac{nb}{r}} \log n)$ days.*

¹ The notion of (r, b) -division is formally defined in Section 3.1

In Section 3.1 we obtain a stronger version of the above theorem, which we apply in Section 3.2 to improve the exploration bounds for temporal realisations of graphs of treewidth k and planar graphs. The main technical contribution that allows us to strengthen Theorem 5 is Lemma 6 saying that if two vertex sets S and U in an n -vertex temporal graph \mathcal{G} are such that $|U| \leq |S|$ and in every snapshot of \mathcal{G} for every vertex $u \in U$ there exists a path between u and a vertex in S , then $|S|$ agents starting at the vertices of S (one agent per vertex) can explore vertices in U and return to their original positions in at most $4|S|n$ days.

3.1 Tools

For a graph $G = (V, E)$, we say that a vertex $v \in V$ is *reachable from a vertex* $u \in V$ in G if there is a path from u to v in G . We also say that a subset $S \subseteq V$ *reaches a subset* $U \subseteq V$ in G , if every vertex $u \in U$ is reachable from some vertex in S . For a temporal graph \mathcal{G} and subsets S and U of its vertices, we say that S *always reaches* U in \mathcal{G} , if S reaches U in every snapshot of \mathcal{G} .

► **Lemma 6.** *Let \mathcal{G} be a not necessarily always-connected temporal graph with vertex set V , let S be a subset of V of cardinality s , and let U be a subset of V with $|U| \leq s$. If S always reaches U in \mathcal{G} and the lifetime of \mathcal{G} is at least $4sn$, then s agents starting at the vertices of S (one agent per vertex) can explore vertices in U and return to their original positions.*

Proof. Let $S = \{x_1, x_2, \dots, x_s\}$ and let a_1, a_2, \dots, a_s denote the agents that are initially located at the vertices x_1, x_2, \dots, x_s respectively. For convenience, we assume that every vertex in U holds a token, and we restrict our consideration only to the exploration schedules in which the agents collect all the tokens from the vertices in U and bring them to the agents' original locations. We assume that every agent can carry at most one token at a time. We say that a vertex $u \in U$ is *explored by an agent* a , if a starts at its original position, visits u , takes the token of u , moves back to its original location, where she drops the token. A day on which agent a returns to its original location and drops the token of u will be called a *return day of* a . The assumption that an agent can carry only one token at a time implies that the agent can explore at most one vertex between any two consecutive visits to its original location.

Let $U = \{u_1, u_2, \dots, u_r\}$ and, for every $i \in [r]$, denote by t_i the earliest day by which the agents can explore i vertices in U . We will prove by induction on i that $t_i \leq 2n(s + i - 1)$. As $r \leq s$, the inequality for $i = r$ will imply the lemma.

For the base case $i = 1$, we need to show that at least one vertex in U can be explored by day $2ns$. Since every vertex in U is reachable from a vertex in S in every snapshot, by the pigeonhole principle, vertex u_1 is reachable from some fixed vertex $x \in S$ in at least $2n$ snapshots out of the first $2ns$ snapshots. Hence, by Lemma 1, the agent of x can explore u_1 by day $2ns$.

Let now $1 < i \leq r$ and assume that the agents can explore $i - 1$ vertices by time $t_{i-1} \leq 2n(s + i - 2)$. Suppose, towards a contradiction, that the agents cannot explore i vertices in the first $2n(s + i - 1)$ days. Let us fix a fastest exploration schedule in which the agents explore $i - 1$ vertices in U . Without loss of generality, assume that the vertices explored under this schedule are u_1, u_2, \dots, u_{i-1} . For $k \in [s]$, let ℓ_k be the number of vertices explored by agent a_k in the first $2n(s + i - 1)$ days; note that $\ell_1 + \ell_2 + \dots + \ell_s = i - 1$. Furthermore, we denote by $d_1^{(k)} < d_2^{(k)} < \dots < d_{\ell_k}^{(k)}$ the return days of agent a_k and call $(d_1^{(k)}, d_2^{(k)}, \dots, d_{\ell_k}^{(k)})$ the *vector of return days of* a_k . We also assume, without loss of generality, that the schedule is *minimal* in the sense that there is no schedule in which all the agents explore the same number of vertices in U , and all the agents have the same vectors of return days, except one of the agents, say a_k , that has a vector of return days that is lexicographically smaller than $(d_1^{(k)}, d_2^{(k)}, \dots, d_{\ell_k}^{(k)})$.

Next, for an arbitrary but fixed $k \in [s]$, we will count the number of snapshots in the first $2n(s+i-1)$ days in which vertex u_i is reachable from vertex x_k . We claim that in each of the time intervals $[1, d_1^{(k)} - 1]$, $[d_j^{(k)} + 1, d_{j+1}^{(k)} - 1]$, $j \in [\ell_k - 1]$, $[d_{\ell_k}^{(k)} + 1, 2n(s+i-1)]$ there are at most $2(n-1)$ such snapshots. Indeed, if the interval $[1, d_1^{(k)} - 1]$ or any of the intervals $[d_j^{(k)} + 1, d_{j+1}^{(k)} - 1]$, $j \in [\ell_k - 1]$ would contain $2(n-1)$ snapshots in which vertex u_i is reachable from vertex x_k , then we could amend the schedule of agent a_k by ordering her to explore u_i during this time interval and keeping the schedule the same in the other intervals. This would produce a schedule in which a_k would have a lexicographically smaller vector of return days than $(d_1^{(k)}, d_2^{(k)}, \dots, d_{\ell_k}^{(k)})$, contradicting the minimality of the schedule. Also, if the last interval $[d_{\ell_k}^{(k)} + 1, 2n(s+i-1)]$ would contain $2(n-1)$ snapshots in which vertex u_i is reachable from vertex x_k , then agent a_k could explore u_i in this interval, contradicting the assumption that the agents cannot explore i vertices in the first $2n(s+i-1)$ days. Hence the total number of snapshots in which u_i is reachable from vertex x_k in the first $2n(s+i-1)$ snapshots is at most $2(n-1)(\ell_k + 1) + \ell_k$. Consequently, the total number of snapshots in which u_i is reachable from any vertex in S in the first $2n(s+i-1)$ snapshots is at most

$$\sum_{k=1}^s (2(n-1)(\ell_k + 1) + \ell_k) = 2(n-1)(s+i-1) + i - 1 < 2n(s+i-1),$$

which contradicts the assumption that S always reaches U in \mathcal{G} . \blacktriangleleft

We will now use Lemma 6 to prove a stronger version of Theorem 5. The notion of (r, b) -division was introduced by Erlebach et al. [3] and it generalizes the notion of r -divisions used by Frederickson [6]. For positive integers r and b (which might be functions of n), a (r, b) -division of a graph $G = (V, E)$ with n vertices is given by a set $S \subseteq V$ and a partition of $G[V \setminus S]$ into $O(n/r)$ (not necessarily connected) components, each associated with a boundary set consisting of vertices from S , such that the following properties hold:

- (1) Each component contains at most r vertices.
- (2) The boundary set of each component has size at most b .
- (3) The boundary sets of different components may overlap, and the union of the boundary sets of all components is S .
- (4) Every edge of G that has only one endpoint in a component has its other endpoint in the boundary set of that component.

► **Theorem 7.** *A temporal graph \mathcal{G} , whose underlying graph has a (r, b) -division, can be explored in $O((n + \max\{r, b\})(r + b)) \frac{nb}{r} \log n$ days.*

Proof. We will use b agents to explore all $O(n/r)$ components one by one. Consider the exploration of a component C and its boundary set B . Since the graph is always-connected, the definition of (r, b) -division implies that B always reaches C in $\mathcal{G}[B \cup C]$, which allows us to apply Lemma 6 as follows. First, using Lemma 1, we position at most b agents at the boundary vertices in at most $n-1$ days. Next, we partition $|C|$ into $\lfloor |C|/|B| \rfloor$ subsets, each with $|B|$ elements, and the subset of the $|C| - |B| \lfloor |C|/|B| \rfloor$ remaining elements. By Lemma 6, any of these subsets can be explored in $4|B|(|C| + |B|)$ days in $\mathcal{G}[B \cup C]$. Therefore, C can be explored in $O((\lfloor |C|/|B| \rfloor + 1)|B|(|C| + |B|)) = O(\max\{r, b\}(r + b))$ days, and the set $B \cup C$ in $O(n + \max\{r, b\}(r + b))$ days. Consequently, the entire graph \mathcal{G} can be explored in $O((n + \max\{r, b\}(r + b)) \frac{n}{r})$ days using b agents, and hence, by Lemma 2, it can be explored in $O((n + \max\{r, b\}(r + b)) \frac{nb}{r} \log n)$ days with a single agent. \blacktriangleleft

3.2 Applications

3.2.1 Bounded treewidth graphs

It was shown in [3] that temporal graphs whose underlying graph has treewidth at most k can be explored in $O(k^{1.5}n^{1.5} \log n)$ days. This bound provides an improvement over the general $O(n^2)$ bound whenever $k = o\left(n^{1/3}/\log^{2/3} n\right)$. A key ingredient of the proof of this result was the fact that graphs with treewidth at most k admit a $(2\sqrt{n/k}, 6k)$ -division (see Lemma 4.4 in [3]). Using exactly the same proof as in [3], but replacing $\sqrt{\frac{n}{k}}$ and \sqrt{nk} with \sqrt{n} everywhere, one can obtain the following

► **Lemma 8** (adaptation of Lemma 4.4 [3]). *Any graph of treewidth at most k admits a $(2\sqrt{n}, 6k)$ -division.*

We will use this latter fact together with Theorem 7 to derive an improved exploration bound for graphs of treewidth at most k .

► **Theorem 9.** *An n -vertex temporal graph, whose underlying graph has treewidth at most k , can be explored in $O(kn^{1.5} \log n)$ days.*

Proof. We can assume, without loss of generality, that $k = o(n^{0.5})$, as otherwise the bound in the statement becomes $\omega(n^2)$ and the result clearly holds. Under this assumption, Lemma 8 and Theorem 7 imply that an n -vertex temporal graph, whose underlying graph has treewidth at most k , can be explored in $O\left((n + \sqrt{n}(\sqrt{n} + k))\sqrt{nk} \log n\right) = O(kn^{1.5} \log n)$ days. ◀

We note that Theorem 9 improves the previous bound from [3] as well as implies an improvement over the general $O(n^2)$ bound for graphs with treewidth $k = o(n^{0.5}/\log n)$.

3.2.2 Planar graphs

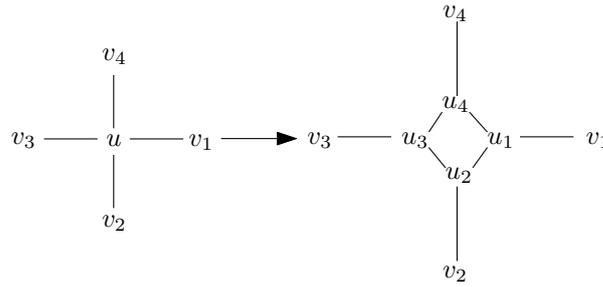
It was shown in [3] that temporal realisations of planar graphs can be explored in $O(n^{1.8} \log n)$ days. We will follow a similar strategy as in [3] but use our Theorem 7 to reduce the bound to $O(n^{1.75} \log n)$.

► **Theorem 10.** *An n -vertex temporal graph, whose underlying graph is planar, can be explored in $O(n^{1.75} \log n)$ days.*

Proof. Frederickson proved that planar graphs admit $(r, O(\sqrt{r}))$ -divisions for any $1 \leq r \leq n$ [6]. Applying this result with $r = \sqrt{n}$ and Theorem 7 we conclude that a temporal realisation of an n -vertex planar graph can be explored in $O\left((n + r(r + \sqrt{r}))\frac{n}{\sqrt{r}} \log n\right) = O\left((n^2/\sqrt{r} + r^{1.5}n) \log n\right) = O(n^{1.75} \log n)$ days. ◀

4 Subcubic planar graphs

Temporal realisations of graphs of maximum degree at most 2 can be explored in linear time. Indeed, a connected graph of maximum degree 2 is either a path or a cycle. Temporal realisations of paths are trivially explorable in linear time, because every snapshot in such temporal graphs must be the same connected graph. As shown in [3], temporal realisations of cycles are also explorable in linear time. On the other hand, it was proved in [3] that some temporal realisations of planar graphs of maximum degree 4 cannot be explored faster than in $\Omega(n \log n)$ days even if every snapshot is a path.



■ **Figure 1** An illustration of the transformation from the neighbourhood around the vertex v in G to the 4-cycle (u_1, u_2, u_3, u_4) in H .

► **Theorem 11** (Theorem 4.1, [3]). *There exist temporal realisations of n -vertex planar graphs of maximum degree 4, in which every snapshot is a path, that cannot be explored faster than $\Omega(n \log n)$ days.*

This leaves an intriguing open case of exploration time of temporal realisations of planar graphs of maximum degree 3. In particular, are such temporal graphs always explorable in linear time? As we shall see below, in general, such temporal graphs can require $\Omega(n \log n)$ days for their exploration. To prove the result, we will transform the construction from the proof of Theorem 11 and apply the following edge contraction lemma from [3].

► **Lemma 12** (edge contraction, Lemma 2.4, [3]). *Let G be a graph such that every temporal realisation of G with lifetime at least t can be explored in t days. Let G' be a graph that is obtained from G by contracting edges. Then every temporal realisation of G' with lifetime t can also be explored in t days.*

► **Theorem 13.** *There exist temporal realisations of n -vertex subcubic planar graphs that cannot be explored faster than $\Omega(n \log n)$ days.*

Proof. Let $n' \geq 16$, let G be an n' -vertex planar graph of maximum degree 4, and let \mathcal{G} be a temporal realisation of G for which every exploration requires at least $cn' \log n'$ days for some positive constant c . Such G and \mathcal{G} exist by Theorem 11.

From graph G we obtain a graph H as follows. For every vertex u in G with 4 neighbours v_1, v_2, v_3, v_4 , we delete u from G , add 4 new vertices u_1, u_2, u_3, u_4 , forming a 4-cycle (u_1, u_2, u_3, u_4) , and add 4 new edges $\{u_1, v_1\}, \{u_2, v_2\}, \{u_3, v_3\}, \{u_4, v_4\}$ (see Figure 1). Let n be the number of vertices in H . Clearly, $n' \leq n \leq 4n'$ and G is obtained from H by contracting edges. Furthermore, it is not hard to see that H is planar and every vertex in H has degree at most 3. Therefore, there exists a temporal realisation \mathcal{H} of H for which every exploration requires at least $cn' \log n'$ days. Indeed, otherwise, by Lemma 12, \mathcal{G} could be explored in less than $cn' \log n'$ days, which would contradict our assumption. Hence, \mathcal{H} cannot be explored in less than

$$cn' \log n' \geq c \frac{n}{4} \log \frac{n}{4} \geq \frac{c}{8} n \log n = \Omega(n \log n)$$

days, where the latter inequality uses the assumption that $n \geq 16$. ◀

► **Remark.** We note that in the temporal graph \mathcal{G} from the proof of Theorem 11 every snapshot is a path. The transformation in the above proof of Theorem 13 can be easily specified in such a way that every snapshot in \mathcal{H} is also a path. We leave the details to the interested reader.

References

- 1 Eleni C Akrida, George B Mertzios, Paul G Spirakis, and Christoforos Raptopoulos. The temporal explorer who returns to the base. *Journal of Computer and System Sciences*, 120:179–193, 2021.
- 2 Hans L Bodlaender and Tom C van der Zanden. On exploring always-connected temporal graphs of small pathwidth. *Information Processing Letters*, 142:68–71, 2019.
- 3 Thomas Erlebach, Michael Hoffmann, and Michael Kammer. On temporal graph exploration. *Journal of Computer and System Sciences*, 119:1–18, 2021.
- 4 Thomas Erlebach, Frank Kammer, Kelin Luo, Andrej Sajenko, and Jakob T Spooner. Two moves per time step make a difference. In *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
- 5 Thomas Erlebach and Jakob T. Spooner. Exploration of k-edge-deficient temporal graphs. In Anna Lubiw and Mohammad Salavatipour, editors, *Algorithms and Data Structures*, pages 371–384, Cham, 2021. Springer International Publishing.
- 6 Greg Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal of Computing*, 16:1004–1022, 1987.
- 7 Othon Michail and Paul G Spirakis. Traveling salesman problems in temporal graphs. *Theoretical Computer Science*, 634:1–23, 2016.
- 8 Shadi Taghian Alamouti. Exploring temporal cycles and grids. Master’s thesis, Concordia University, 2020.

Building Squares with Optimal State Complexity in Restricted Active Self-Assembly

Robert M. Alaniz ✉

Department of Computer Science,
University of Texas Rio Grande Valley, TX, USA

Sonya C. Cirlos ✉

Department of Computer Science,
University of Texas Rio Grande Valley, TX, USA

Elise Grizzell ✉

Department of Computer Science,
University of Texas Rio Grande Valley, TX, USA

Robert Schweller ✉

Department of Computer Science,
University of Texas Rio Grande Valley, TX, USA

Tim Wylie ✉

Department of Computer Science,
University of Texas Rio Grande Valley, TX, USA

David Caballero ✉

Department of Computer Science,
University of Texas Rio Grande Valley, TX, USA

Timothy Gomez ✉

Department of Computer Science,
University of Texas Rio Grande Valley, TX, USA

Andrew Rodriguez ✉

Department of Computer Science,
University of Texas Rio Grande Valley, TX, USA

Armando Tenorio ✉

Department of Computer Science,
University of Texas Rio Grande Valley, TX, USA

Abstract

Tile Automata is a recently defined model of self-assembly that borrows many concepts from cellular automata to create active self-assembling systems where changes may be occurring within an assembly without requiring attachment. This model has been shown to be powerful, but many fundamental questions have yet to be explored. Here, we study the state complexity of assembling $n \times n$ squares in seeded Tile Automata systems where growth starts from a seed and tiles may attach one at a time, similar to the abstract Tile Assembly Model. We provide optimal bounds for three classes of seeded Tile Automata systems (all without detachment), which vary in the amount of complexity allowed in the transition rules. We show that, in general, seeded Tile Automata systems require $\Theta(\log^{\frac{1}{4}} n)$ states. For Single-Transition systems, where only one state may change in a transition rule, we show a bound of $\Theta(\log^{\frac{1}{3}} n)$, and for deterministic systems, where each pair of states may only have one associated transition rule, a bound of $\Theta(\frac{\log n}{\log \log n})^{\frac{1}{2}}$.

2012 ACM Subject Classification Theory of computation \rightarrow Computational geometry; Applied computing \rightarrow Computational biology; Theory of computation \rightarrow Self-organization

Keywords and phrases Active Self-Assembly, State Complexity, Tile Automata

Digital Object Identifier 10.4230/LIPIcs.SAND.2022.6

Supplementary Material *Software (Source Code)*: <https://github.com/asarg/AutoTile>
archived at `swh:1:dir:fd83de54cc0e347b80c90911b19bd8e0266a5bc8`

Funding This research was supported in part by National Science Foundation Grant CCF-1817602.

Acknowledgements We would like to thank the reviewers for their comments, specifically for pointing us toward relevant Cellular Automata Literature.

1 Introduction

Self-assembly is the process by which simple elements in a system organize themselves into more complex structures based on a set of rules that govern their interactions. These types of systems occur naturally and can be easily constructed artificially to offer many advantages when building micro or nanoscale objects. One abstraction of these systems that has yielded interesting results is Tile Self-Assembly.



© Robert M. Alaniz, David Caballero, Sonya C. Cirlos, Timothy Gomez, Elise Grizzell, Andrew Rodriguez, Robert Schweller, Armando Tenorio, and Tim Wylie;
licensed under Creative Commons License CC-BY 4.0

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).

Editors: James Aspnes and Othon Michail; Article No. 6; pp. 6:1–6:18



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In the abstract Tile Assembly Model (aTAM) [35], the elements of a system are represented using labeled unit squares called tiles. A system is initialized with a seed (a tile or assembly) that grows as other single tiles attach until there are no more valid attachments. The behavior of a system can then be programmed, using the interactions of tiles, and is known to be capable of Turing Computation [35], is Intrinsically Universal [14], and can assemble general scaled shapes [33]. However, many of these results utilize a concept called *cooperative binding*, where a tile must attach to an assembly using the interaction from two other tiles. Unlike with cooperative binding, the non-cooperative aTAM is not Intrinsically Universal [25, 27] and more recent work has shown that it is not capable of Turing Computation [26]. Many extensions of this model increase the power of non-cooperative systems [4, 16, 18, 22, 23, 30].

One recent model of self-assembly is Tile Automata [8]. This model marries the concept of state changes from Cellular Automata [19, 28, 37] and the assembly process from the 2-Handed Assembly model (2HAM) [6]. Previous work [3, 7, 8] has explored Tile Automata as a unifying model for comparing the relative powers of the many different Tile Assembly models. The complexity of verifying the behavior of systems along with their computational power was studied in [5]. Many of these works impose additional experimentally motivated limitations on the Tile Automata model that help connect the model and its capabilities to potential molecular implementations, such as using DNA assemblies with sensors to assemble larger structures [21], building spacial localized circuits on DNA origami [10], or DNA walkers that sort cargo [34].

In this paper, we explore the aTAM generalized with state changes; we define our producible assemblies as what can be grown by attaching tiles one at a time to a seed tile or performing transition rules, which we refer to as seeded Tile Automata. This is a bounded version of Asynchronous Cellular Automata [15]. Reachability problems, which are similar to verification problems in self-assembly, have been studied with many completeness results [13]. Further, the freezing property used in this and previous work also exists in Cellular Automata [20, 29].¹ Freezing is defined differently in Cellular Automata by requiring that there exists an ordering to the states.

While Tile Automata has many possible metrics, we focus on the number of states needed to uniquely assemble $n \times n$ squares at the smallest constant temperature, $\tau = 1$. We achieve optimal bounds in three versions of the model with varying restrictions on the transition rules. Our results, along with previous results in the aTAM, are outlined in Table 1.

1.1 Previous Work

In the aTAM, the number of tile types needed, for nearly all n , to construct an $n \times n$ square is $\Theta(\frac{\log n}{\log n \log n})$ [1, 31] with temperature $\tau = 2$ (row 2 of Table 1). The same lower bounds hold for $\tau = 1$ (row 1 of Table 1). The run time of this system was also shown to be optimal $\Theta(n)$ [1]. Other bounds for building rectangles were shown in [2]. While no tighter bounds² have been shown for $n \times n$ squares at $\tau = 1$ in the aTAM, generalizations to the model that allow (just-barely) 3D growth have shown an upper bound of $\mathcal{O}(\log n)$ for tile types needed [11]. Recent work in [17] shows improved upper and lower bounds on building thin rectangles in the case of $\tau = 1$ and in (just-barely) 3D.

Other models of self-assembly have also been shown to have a smaller tile complexity, such as the staged assembly model [9, 12] and temperature programming [24]. Investigation into different active self-assembly models have also explored the run time of systems [32, 36].

¹ We would like to thank a reviewer for bringing these works to our attention.

² Other than trivial $\mathcal{O}(n)$ bounds.

■ **Table 1** Bounds on the number of states for $n \times n$ squares in the Abstract Tile Assembly model, with and without cooperative binding, and the seeded Tile Automata model with our transition rules. ST stands for Single-Transition.

Model	τ	$n \times n$ Squares		
		Lower	Upper	Theorem
aTAM	1	$\Omega(\frac{\log n}{\log \log n})$	$\mathcal{O}(n)$	[31], [1]
aTAM	2	$\Theta(\frac{\log n}{\log \log n})$		[31], [1]
Flexible Glue aTAM	2	$\Theta(\log^{\frac{1}{2}} n)$		[2]
Seeded TA Det.	1	$\Theta((\frac{\log n}{\log \log n})^{\frac{1}{2}})$		Thm. 2, 12
Seeded TA ST	1	$\Theta(\log^{\frac{1}{3}} n)$		Thm. 4, 12
Seeded TA	1	$\Theta(\log^{\frac{1}{4}} n)$		Thm. 3, 12

1.2 Our Contributions

In this work, we explore building an important benchmark shape, squares, in non-cooperative seeded Tile Automata. We also consider only affinity-strengthening transition rules that remove the ability for an assembly to break apart. Our results are shown in Table 1.

We start in Section 3 by proving lower bounds for building $n \times n$ squares based on three different transition rule restrictions. The first is nondeterministic or general seeded Tile Automata, where there are no restrictions and a pair of states may have multiple transition rules. The second is Single-Transition rules where only one tile may change states in a transition rule, but we still allow multiple rules for each pair of states. The last restriction, Deterministic, is the most restrictive where each pair of states may only have one transition rule (for each direction).

In Section 4, we use Transition Rules to optimally encode strings in the various versions of the model. We use these encodings as gadgets to seed the future constructions. We show how to build optimal state complexity rectangles in Section 5, and finally optimal state complexity squares in Section 6. Future work is discussed in Section 7.

AutoTile. To test our constructions, we developed AutoTile, a seeded Tile Automata simulator. Each system discussed in the paper is currently available for simulation. AutoTile is available at <https://github.com/asarg/AutoTile>.

2 Definitions

The Tile Automata model differs quite a bit from normal self-assembly models since a *tile* may change *state*, which draws inspiration from Cellular Automata. Thus, there are two aspects of a TA system being: the self-assembling that may occur with tiles in a state and the changes to the states once they have attached to each other. To address these aspects, we define the building blocks and interactions, and then the definitions around the model and what it may assemble or output. Finally, since we are looking at a limited TA system, we also define specific limitations and variations of the model. For reference, an example system is shown in Figure 1.

2.1 Building Blocks

The basic definitions of all self-assembly models include the concepts of tiles, some method of attachment, and the concept of aggregation into larger assemblies. The Cellular Automata aspect also brings in the concept of transitions.

Tiles. Let Σ be a set of *states* or symbols. A tile $t = (\sigma, p)$ is a non-rotatable unit square placed at point $p \in \mathbb{Z}^2$ and has a state of $\sigma \in \Sigma$.

Affinity Function. An *affinity function* Π over a set of states Σ takes an ordered pair of states $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$ and an orientation $d \in D$, where $D = \{\perp, \vdash\}$, and outputs an element of \mathbb{Z}^0 . The orientation d is the relative position to each other with \vdash meaning horizontal and \perp meaning vertical, with the σ_1 being the west or north state respectively. We refer to the output as the *Affinity Strength* between these two states.

Transition Rules. A *Transition Rule* consists of two ordered pairs of states $(\sigma_1, \sigma_2), (\sigma_3, \sigma_4)$ and an orientation $d \in D$, where $D = \{\perp, \vdash\}$. This denotes that if the states (σ_1, σ_2) are next to each other in orientation d (σ_1 as the west/north state) they may be replaced by the states (σ_3, σ_4) .

Assembly. An assembly A is a set of tiles with states in Σ such that for every pair of tiles $t_1 = (\sigma_1, p_1), t_2 = (\sigma_2, p_2)$, $p_1 \neq p_2$. Informally, each position contains at most one tile. Further, we say assemblies are equal in regards to translation. Two assemblies A_1 and A_2 are equal if there exists a vector \vec{v} such that $A_1 = A_2 + \vec{v}$.

Let $B_G(A)$ be the bond graph formed by taking a node for each tile in A and adding an edge between neighboring tiles $t_1 = (\sigma_1, p_1)$ and $t_2 = (\sigma_2, p_2)$ with a weight equal to $\Pi(\sigma_1, \sigma_2)$. We say an assembly A is τ -stable for some $\tau \in \mathbb{Z}^0$ if the minimum cut through $B_G(A)$ is greater than or equal to τ .

2.2 The Tile Automata Model

Here, we define and investigate the *Seeded Tile Automata* model, which differs by only allowing single tile attachments to a growing seed similar to the aTAM.

Seeded Tile Automata. A Seeded Tile Automata system is a 6-tuple $\Gamma = \{\Sigma, \Lambda, \Pi, \Delta, s, \tau\}$ where Σ is a set of states, $\Lambda \subseteq \Sigma$ a set of *initial states*, Π is an *affinity function*, Δ is a set of *transition rules*, s is a stable assembly called the *seed* assembly, and τ is the *temperature* (or threshold). Our results use the most restrictive version of this model where s is a single tile.

Attachment Step. A tile $t = (\sigma, p)$ may attach to an assembly A at temperature τ to build an assembly $A' = A \cup t$ if A' is τ -stable and $\sigma \in \Lambda$. We denote this as $A \rightarrow_{\Lambda, \tau} A'$.

Transition Step. An assembly A is transitionable to an assembly A' if there exists two neighboring tiles $t_1 = (\sigma_1, p_1), t_2 = (\sigma_2, p_2) \in A$ (where t_1 is the west or north tile) such that there exists a transition rule in Δ with the first pair being (σ_1, σ_2) and $A' = (A \setminus \{t_1, t_2\}) \cup \{t_3 = (\sigma_3, p_1), t_4 = (\sigma_4, p_2)\}$. We denote this as $A \rightarrow_{\Delta} A'$.

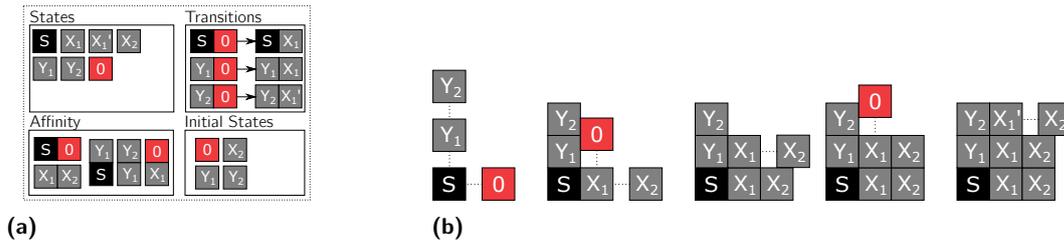


Figure 1 (a) Example of a Tile Automata system, it should be noted that $\tau = 1$ and state S is our seed. (b) A walkthrough of our example Tile Automata system building the 3×3 square it uniquely produces. We use dotted lines throughout our paper to represent tiles attaching to one another.

Producibles. We refer to both attachment steps and transition steps as production steps, we define $A \rightarrow_* A'$ as the transitive closure of $A \rightarrow_{\Lambda, \tau} A'$ and $A \rightarrow_{\Delta} A'$. The set of *producible assemblies* for a Tile Automata system $\Gamma = \{\Sigma, \Lambda, \Pi, \Delta, s, \tau\}$ is written as $PROD(\Gamma)$. We define $PROD(\Gamma)$ recursively as follows,

- $s \in PROD(\Gamma)$
- $A' \in PROD(\Gamma)$ if $\exists A \in PROD(\Gamma)$ such that $A \rightarrow_{\Lambda, \tau} A'$.
- $A' \in PROD(\Gamma)$ if $\exists A \in PROD(\Gamma)$ such that $A \rightarrow_{\Delta} A'$.

Terminal Assemblies. The set of terminal assemblies for a Tile Automata system $\Gamma = \{\Sigma, \Lambda, \Pi, \Delta, \tau\}$ is written as $TERM(\Gamma)$. This is the set of assemblies that cannot grow or transition any further. Formally, an assembly $A \in TERM(\Gamma)$ if $A \in PROD(\Gamma)$ and there does not exist any assembly $A' \in PROD(\Gamma)$ such that $A \rightarrow_{\Lambda, \tau} A'$ or $A \rightarrow_{\Delta} A'$. A Tile Automata system $\Gamma = \{\Sigma, \Lambda, \Pi, \Delta, s, \tau\}$ *uniquely* assembles an assembly A if $A \in TERM(\Gamma)$, and for all $A' \in PROD(\Gamma)$, $A' \rightarrow_* A$.

2.3 Limited Model Reference

We explore an extremely limited version of seeded TA that is affinity-strengthening, freezing, and may be a single-transition system. We investigate both deterministic and non-deterministic versions of this model.

Affinity Strengthening. We only consider transitions rules that are affinity strengthening, meaning for each transition rule $((\sigma_1, \sigma_2), (\sigma_3, \sigma_4), d)$, the bond between (σ_3, σ_4) must be at least the strength of (σ_1, σ_2) . Formally, $\Pi(\sigma_3, \sigma_4, d) \geq \Pi(\sigma_1, \sigma_2, d)$. This ensures that transitions may not induce cuts in the bond graph.

In the case of non-cooperative systems ($\tau = 1$), the affinity strength between states is always 1 so we may refer to the affinity function as an affinity set Λ_s , where each affinity is a 3-pule (σ_1, σ_2, d) .

Freezing. Freezing systems were introduced with Tile Automata. A freezing system simply means that a tile may transition to any state only once. Thus, if a tile is in state A and transitions to another state, it is not allowed to ever transition back to A .

Deterministic vs. Nondeterministic. For clarification, a deterministic system in TA has only one possible production step at a time, whether that be an attachment or a state transition. A nondeterministic system may have many possible production steps and any choice may be taken.

Single-Transition System. We restrict our TA system to only use single-transition rules. This means that for each transition rule one of the states may change, but not both. It should be noted that we still allow Nondeterminism in this system.

3 State Space Lower Bounds

Let $p(n)$ be a function from the positive integers to the set $\{0, 1\}$, informally termed a *proposition*, where 0 denotes the proposition being false and 1 denotes the proposition being true. We say a proposition $p(n)$ holds for *almost all* n if $\lim_{n \rightarrow \infty} \frac{1}{n} \sum_{i=1}^n p(i) = 1$.

► **Lemma 1.** *Let U be a set of TA systems, b be a one-to-one function mapping each element of U to a string of bits, and ϵ a real number from $0 < \epsilon < 1$. Then for almost all integers n , any TA system $\Gamma \in U$ that uniquely assembles either an $n \times n$ square or a $1 \times n$ line has a bit-string of length $|b(\Gamma)| \geq (1 - \epsilon) \log n$.*

Proof. For a given $i \geq 1$, let $M_i \in U$ denote the TA system in U with the minimum value $|b(M_i)|$ over all systems in U that uniquely assembly an $i \times i$ square or $1 \times i$ line, and let M_i be undefined if no such system in U builds such a shape. Let $p(i)$ be the proposition that $|b(M_i)| \geq (1 - \epsilon) \log i$. We show that $\lim_{n \rightarrow \infty} \frac{1}{n} \sum_{i=1}^n p(i) = 1$. Let $R_n = \{M_i | 1 \leq i \leq n, |b(M_i)| < (1 - \epsilon) \log n\}$. Note that $n - |R_n| \leq \sum_{i=1}^n p(i)$. By the pigeon-hole principle, $|R_n| \leq 2^{(1-\epsilon) \log n} = n^{(1-\epsilon)}$. Therefore,

$$\lim_{n \rightarrow \infty} \frac{1}{n} \sum_{i=1}^n p(i) \geq \lim_{n \rightarrow \infty} \frac{1}{n} (n - |R_n|) \geq \lim_{n \rightarrow \infty} \frac{1}{n} (n - n^{1-\epsilon}) = 1. \quad \blacktriangleleft$$

► **Theorem 2 (Deterministic TA).** *For almost all n , any Deterministic Tile Automata system that uniquely assembles either a $1 \times n$ line or an $n \times n$ square contains $\Omega\left(\frac{\log n}{\log \log n}\right)^{\frac{1}{2}}$ states.*

Proof. We can create a one-to-one mapping $b(\Gamma)$ from any deterministic TA system to bit-strings in the following way. Let S denote the set of states in a given system. We encode the state set in $\mathcal{O}(\log |S|)$ bits, we encode the affinity function in a $|S| \times |S|$ table of strengths in $\mathcal{O}(|S|^2)$ bits (assuming a constant bound on bonding thresholds), and we encode the rules of the system in an $|S| \times |S|$ table mapping pairs of rules to their unique new pair of rules using $\mathcal{O}(|S|^2 \log |S|)$ bits, for a total of $\mathcal{O}(|S|^2 \log |S|)$ bits to encode any $|S|$ state system.

Let Γ_n denote the smallest state system that uniquely assembles an $n \times n$ square (or similarly a $1 \times n$ line), and let S_n denote the state set. By Lemma 1, $|b(\Gamma_n)| \geq (1 - \epsilon) \log n$ for almost all n , and so $|S_n|^2 \log |S_n| = \Omega(\log n)$ for almost all n . We know that $|S_n| = \mathcal{O}(\log n)$, so for some constant c , $|S_n| \geq c \left(\frac{\log n}{\log \log n}\right)^{\frac{1}{2}}$ for almost all n . ◀

► **Theorem 3 (Nondeterministic TA).** *For almost all n , any Tile Automata system (in particular any Nondeterministic system) that uniquely assembles either a $1 \times n$ line or an $n \times n$ square contains $\Omega(\log^{\frac{1}{4}} n)$ states.*

► **Theorem 4 (Single-Transition TA).** *For almost all n , any Single-Transition Tile Automata system that uniquely assembles either a $1 \times n$ line or an $n \times n$ square contains $\Omega(\log^{\frac{1}{3}} n)$ states.*

4 String Unpacking

A key tool in our constructions is the ability to build strings efficiently. We do so by encoding the string in the transition rules.

► **Definition 5** (String Representation). *An assembly A over states Σ represent a string S over a set of symbols U if there exists a mapping from the elements of U to the elements of Σ and a $1 \times |S|$ (or $|S| \times 1$) subassembly $A' \sqsubset A$, such that the state of the i^{th} tile of A' maps to the i^{th} symbol of S for all $0 \leq i \leq |S|$.*

4.1 Deterministic Transitions

We start by showing how to encode a binary string of length n in a set of (freezing) transition rules that take place on a $2 \times (n + 2)$ rectangle that will print the string on its right side. We extend this construction to work for an arbitrary base string.

4.1.1 Overview

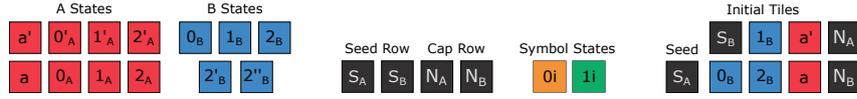
Consider a system that builds a length n string. First, we create a rectangle of index states that is two wide as seen on the left side of Figure 5c. Each row has a unique pair of index states so each bit of the string is uniquely indexed. We divide the index states into two groups based on which column they are in, and which “digit” they represent. Let $r = \lceil n^{\frac{1}{2}} \rceil$. Starting with index states A_0 and B_0 , we build a counter pattern with base r . We use $\mathcal{O}(n^{\frac{1}{2}})$ states shown in Figure 2 to build this pattern. We encode each bit of the string in a transition rule between the two states that index that bit. A table with these transition rules can be seen in Figure 5b.

The pattern is built in r sections of size $2 \times r$ with the first section growing off of the seed. The tile in state S_A is the seed. There is also a state S_B that has affinity for the right side of S_A . The building process is defined in the following steps for each section.

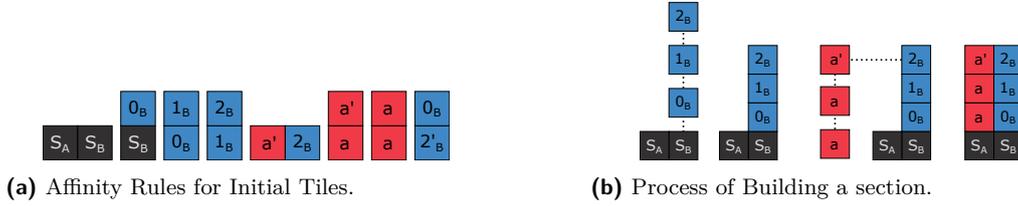
1. The states $S_B, 0_B, 1_B, \dots, (r - 1)_B$ grow off of S_B , forming the right column of the section. The last B state allows for a' to attach on its west side. a tiles attach below a' and below itself. This places a states in a row south toward the state S_A , depicted in Figure 3b.
2. Once a section is built, the states begin to follow their transition rules shown in Figure 4a. The a state transitions with seed state S_A to begin indexing the A column by changing state a to state 0_A . For $1 \leq y \leq n - 2$, state a vertically transitions with the other y'_A states, incrementing the index by changing from state a to state $(y + 1)_A$.
3. This new index state z_A propagates up by transitioning the a tiles to the state z_A as well. Once the z_A state reaches a' at the top of the column, it transitions a' to the state z'_A . Figure 4b presents this process of indexing the A column.
4. If $z < n - 1$, there is a horizontal transition rule from states $(z'_A, n - 1_B)$ to states $(z'_A, n - 1'_B)$. The state 0_B attaches to the north of $n - 1_B$ and starts the next section. If $z = n$, there does not exist a transition.
5. This creates an assembly with a unique state pair in each row as seen in the first column of Figure 5c.

4.1.2 States

An example system with the states required to print a length-9 string are shown in Figure 2. The first states build the seed row of the assembly. The seed tile has the state S_A with initial tiles in state S_B . The index states are divided into two groups. The first set of index states, which we call the A index states, are used to build the left column. For each i , $0 \leq i < r$, we have the states i_A and i'_A . There are two states a and a' , which exist as initial tiles and act as “blank” states that can transition to the other A states. The second set of index states are the B states. Again, we have r B states numbered from 0 to $r - 1$, however, we do not



■ **Figure 2** States to build a length-9 string in deterministic Tile Automata.



■ **Figure 3** (a) Affinity rules to build each section. We only show affinity rules that are actually used in our system for initial tiles to attach, while our system would have more rules in order to meet the affinity strengthening restriction. (b) The B column attaches above the state S_B as shown by the dotted lines. The a' attaches to the left of 2_B and the other a states may attach below it until they reach S_A .

have a prime for each state. Instead, there are two states $r - 1'_B$ and $r - 1''_B$, that are used to control the growth of the next column and the printing of the strings. The last states are the symbol states 0_S and 1_S , the states that represent the string.

4.1.3 Affinity Rules/Placing Section

Here, we describe the affinity rules for building the first section. We later describe how this is generalized to the other $r - 1$ sections. We walk through this process in Figure 3b. To begin, the B states attach in sequence above the tile S_B in the seed row. Assuming $r^2 = n$, n is a perfect square, the first state to attach is 0_B . 1_B attaches above this tile and so on. The last B state $r - 1_B$ does not have affinity with 0_B , so the column stops growing. However, the state a' has affinity on the left of $r - 1_B$ and can attach. a has affinity for the south side of a' , so it attaches below. The a state also has a vertical affinity with itself. This grows the A column southward toward the seed row.

If n is not a perfect square, we start the index state pattern at a different value. We do so by finding the value $q = r^2 - n$. In general, the state i_B attaches above S_B for $i = q\%r$.

4.1.4 Transition Rules/Indexing A column

Once the A column is complete and the last A state is placed above the seed, it transitions with S_A to 0_A (assuming $r^2 = n$). A has a vertical transition rule with i_A ($0 \leq i < r$) changing the state A to state i_A . This can be seen in Figure 4a, where the 0_A state is propagated upward to the A' state. The A' state also transitions when 0_A is below it, going from state A' to state $0'_A$. If n is not a perfect square, then A transitions to i_A for $i = \lfloor q/r \rfloor$.

Once the transition rules have finished indexing the A column if $i < r - 1$, the last state i'_A transitions with $r - 1_B$ changing the state $r - 1_B$ to $r - 1'_B$. This transition can be seen in Figure 4b. The new state $r - 1'_B$ has an affinity rule allowing 0_B to attach above it allowing the next section to be built. When the state A is above a state j'_A , $0 \leq j < r - 1$, it transitions with that state changing from state A to $j + 1_A$, which increments the A index.

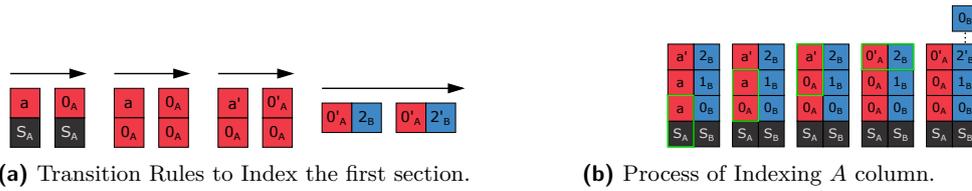


Figure 4 (a) The first transition rule used is takes place between the seed S_A and the a state changing to 0_A . The state 0_A changes the states north of it to 0_A or $0'_A$. Finally, the state $0'_A$ transitions with 2_B (b) Once the a states reach the seed row they transition with the state S_A to go to 0_A . This state propagates upward to the top of the section.

4.1.5 Look up

After creating a $2 \times (n + 2)$ rectangle, we can encode a length n string S into the transitions rules. Note that each row of our assembly consists of a unique pair of index states, which we call a *bit gadget*. Each bit gadget will *look up* a specific bit of our string and transition the B tile to a state representing the value of that bit.

Figure 5b shows how to encode a string S in a table with two columns using r digits to index each bit. From this encoding, we create our transition rules. Consider the k^{th} bit of S (where the 0^{th} bit is the least significant bit) for $k = ir + j$. Add transition rules between the states i_A and j_B , changing the state j_B to either 0_S or 1_S based on the k^{th} bit of S . This transition rule is slightly different for the northmost row of each section as the state in the A column is i'_A . Also, we do not want the state in the B column, $r - 1_B$, to prematurely transition to a symbol state. Thus, we have the two states $r - 1'_B$ and $r - 1''_B$. As mentioned, once the A column finishes indexing, it changes the state $r - 1_B$ to state $r - 1'_B$, allowing for 0_B to attach above it, which starts the next column. Once the state 0_B (or a symbol state) is above $r - 1'_B$, there are no longer any possible undesired attachments, so the state transitions to $r - 1''_B$, which has the transition to the symbol state.

The last section has a slightly different process as $r - 1_B$ state will never have a 0_B attach above it, so we have a different transition rule. This alternate process is shown in Figure 5a. The state $r - 1'_A$ has a vertical affinity with the cap state N_A . This state allows N_B to attach on its right side. This state transitions with $r - 1_B$ below it, changing it directly to $r - 1''_B$, allowing the symbol state to print.

► **Theorem 6.** *For any binary string s with length $n > 0$, there exists a freezing tile automata system Γ_s with deterministic transition rules, that uniquely assembles an $2 \times (n + 2)$ assembly A_S that represents S with $\mathcal{O}(n^{\frac{1}{2}})$ states.*

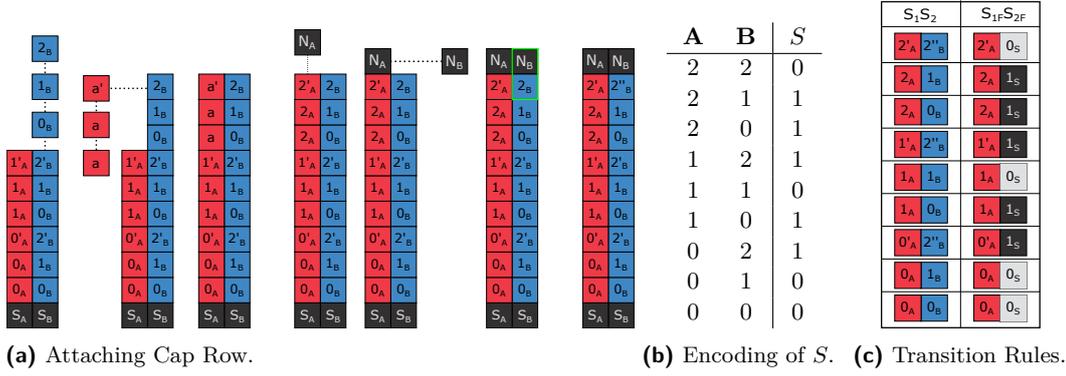
4.1.6 Arbitrary Base

In order to optimally build rectangles, we first print arbitrary base strings. Here, we show how to generalize Theorem 6 to print base- b strings.

► **Corollary 7.** *For any base- b string S with length $n > 0$, there exists a freezing tile automata system Γ with deterministic transition rules, that uniquely assembles an $(n + 2) \times 2$ assembly which represents S with $\mathcal{O}(n^{\frac{1}{2}} + b)$ states.*

4.2 Nondeterministic Single-Transition Systems

For the case of Single-Transition systems, we use the same method from above but instead building bit gadgets that are of size 3×2 . Expanding to 3 columns allows for a third index digit to be used giving us an upper bound of $\mathcal{O}(n^{\frac{1}{3}})$. The second row will be used for error



■ **Figure 5** (a) Once the last section finishes building the state N_A attaches above $2'_A$. N_B then attaches to the assembly and transitions with 2_B changing it directly to $2''_B$ so the string may begin printing. (b) A table indexing the string $S = 011101100$ using two columns and base $|S|^{\frac{1}{2}}$. (c) Transition Rules to print S . We build an assembly where each row has a unique pair of index states in ascending order.

checking which we will describe later in the section. This system utilizes Nondeterministic transitions, (two states may have multiple rules with the same orientation) and is non-freezing (a tile may repeat states). This system also contains cycles in its production graph, this implies the system may run indefinitely. We conjecture this system has a polynomial run time. Here, let $r = \lceil n^{\frac{1}{3}} \rceil$.

4.2.1 Index States and Look Up States

We generalize the method from above to start from a C column. The B column now behaves as the second index of the pattern and is built using B' and B as the A column was in the previous system. Once the B reaches the seed row, it is indexed with its starting value. This construction also requires bit gadgets of height 2, so we will use index states i_A, i_B, i_C and north index states i_{Au}, i_{Bu}, i_{Cu} for $0 \leq i < r$. This allows us to separate the two functions of the bit gadget into each row. The north row has transition rules to control the building of each section. The bottom row has transition rules that encode the represented bit.

In addition to the index states, we use $2r$ look up states, 0_{Ci} and 1_{Ci} for $0 \leq i < r$. These states are used as intermediate states during the look up. The first number (0 or 1) represents the value of the retrieved bit, while the second number represents the C index of the bit. The A and B indices of the bit will be represented by the other states in the transition rule.

In the same way as the previous construction, we build the rightmost column first. We include the C index states as initial states and allow 0_C to attach above S_C . We include affinity rules to build the column northwards as follows starting with the southmost state $0_C, 0_{Cu}, 1_C, 1_{Cu}, \dots, r - 2_{Cu}, r - 1_C, r - 1_{Cu}$.

To build the other columns, the state b' can attach on the left of $r - 1_{Cu}$. The state b is an initial state and attaches below b' and itself to grow downward toward the seed row. The state b transitions with the seed row as in the previous construction to start the column. However, we alternate between C states and Cu states. The state b above i_C transitions b to i_{Cu} . If b is above i_{Cu} it transitions to i_C . The state b' above state i_B transitions to i'_{Bu} . If $i < r - 1$, the state i'_B and $r - 1_{Cu}$ transition horizontally changing $r - 1'_{Cu}$, which allows 0_C to attach above it to repeat the process. This is shown in Figure 6b.

5.3 Walls and Stopping

The computation of a column is complete when a no-carry tile is placed next to any tile with a north state. The transition rule changes the no-carry tile into a north state, preventing the column from growing any higher. The tiles in the column with a carry transition to remove the carry information, as it is no longer needed for computation. A tile with a carry changes states into a state without the carry. The next column can begin computation when the plus tile transitions into a south tile, thus allowing a new plus tile to be attached. The assembly stops growing to the right when the last column gets stuck in an unfinished state. This column, the stopping column, has carry information in every tile that is unable to transition. When a carry tile is placed next to a north tile, there is no transition rule to change the state of the carry tile, thus preventing any more growth to the right of the column.

► **Theorem 10.** *For all $n > 0$, there exists a Tile Automata system that uniquely assembles a $\mathcal{O}(\log n) \times n$ rectangle using,*

- *Deterministic Transition Rules and $\mathcal{O}(\log^{\frac{1}{2}} n)$ states.*
- *Single-Transition Transition Rules and $\Theta(\log^{\frac{1}{3}} n)$ states.*
- *Nondeterministic Transition Rules and $\Theta(\log^{\frac{1}{4}} n)$ states.*

5.4 Arbitrary Bases

Here, we generalize the binary counter process for arbitrary bases. The basic functionality remains the same. The digits of the number are computed one at a time going up the column. If a digit has a carry, then a carry tile attaches to the north, just like the binary counter. If a digit has no carry, then a no-carry tile is attached to the north. The half adder addition step still adds the newly placed carry or no-carry tile with the west neighbor to compute the next digit. This requires adding $\mathcal{O}(b)$ counter states to the system, where b is the base.

► **Theorem 11.** *For all $n > 0$, there exists a Deterministic Tile Automata system that uniquely assembles a $\mathcal{O}(\frac{\log n}{\log \log n}) \times n$ rectangle using $\Theta\left(\left(\frac{\log n}{\log \log n}\right)^{\frac{1}{2}}\right)$ states.*

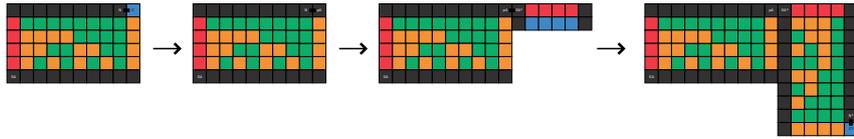
6 Squares

In this section we utilize the rectangle constructions to build $n \times n$ squares using the optimal number of states.

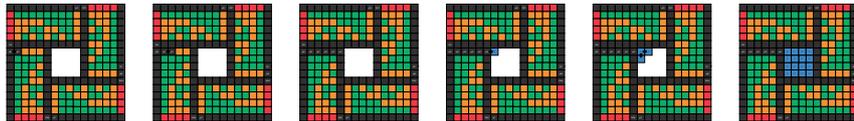
Let $n' = n - 4\lceil \frac{\log n}{\log \log n} \rceil - 2$, and Γ_0 be a deterministic Tile Automata system that builds a $n' \times (4\lceil \frac{\log n}{\log \log n} \rceil + 2)$ rectangle using the process described in Theorem 11. Let Γ_1 be a copy of Γ_0 with the affinity and transition rules rotated 90 degrees clockwise, and the state labels appended with the symbol “*1”. This system will have distinct states from Γ_0 , and will build an equivalent rectangle rotated 90 degrees clockwise. We create two more copies of Γ_0 (Γ_2 and Γ_3), and rotate them 180 and 270 degrees, respectively. We append the state labels of Γ_2 and Γ_3 in a similar way.

We utilize the four systems described above to build a hollow border consisting of the four rectangles, and then adding additional initial states which fill in this border, creating the $n \times n$ square.

We create Γ_n , starting with system Γ_0 , and adding all the states, initial states, affinity rules, and transition rules from the other systems ($\Gamma_1, \Gamma_2, \Gamma_3$). The seed states of the other systems are added as initial states to Γ_n . We add a constant number of additional states and transition rules so that the completion of one rectangle allows for the “seeding” of the next.



■ **Figure 10** The transitions that take place after the first rectangle is built. The carry state transitions to a new state that allows a seed row for the second rectangle to begin growth.



■ **Figure 11** Once all 4 sides of the square build the pD state propagates to the center and allows the light blue tiles to fill in.

Reseeding the Next Rectangle. To Γ_n we add transition rules such that once the first rectangle (originally built by Γ_0) has built to its final width, a tile on the rightmost column of the rectangle will transition to a new state pA . pA has affinity with the state $S_A * 1$, which originally was the seed state of Γ_1 . This allows state $S_A * 1$ to attach to the right side of the rectangle, “seeding” Γ_1 and allowing the next rectangle to assemble (Figure 10). The same technique is used to seed Γ_2 and Γ_3 .

Filler Tiles. When the construction of the final rectangle (of Γ_3) completes, transition rules propagate a state pD towards the center of the square (Figure 11). Additionally, we add an initial state r , which has affinity with itself in every orientation, as well as with state pD on its west side. This allows the center of the square to be filled with tiles.

► **Theorem 12.** *For all $n > 0$, there exists a Tile Automata system that uniquely assembles an $n \times n$ square with,*

- *Deterministic transition rules and $\Theta\left(\left(\frac{\log n}{\log \log n}\right)^{\frac{1}{2}}\right)$ states.*
- *Single-Transition rules and $\Theta(\log^{\frac{1}{3}} n)$ states.*
- *Nondeterministic transition rules and $\Theta(\log^{\frac{1}{4}} n)$ states.*

7 Future Work

This paper showed optimal bounds for uniquely building $n \times n$ squares in three variants of seeded Tile Automata without cooperative binding. En route, we proved upper bounds for constructing strings and rectangles. Serving as a preliminary investigation into constructing shapes in this model. This leaves many open questions:

- As shown in [5], even 1D Tile Automata systems can perform Turing computation. This behavior may imply interesting results for constructing $1 \times n$ lines. We conjecture, it is possible to achieve the optimal bound of $\Theta\left(\left(\frac{\log n}{\log \log n}\right)^{\frac{1}{2}}\right)$ with deterministic rules.
- Our rectangles had a height bounded by $\mathcal{O}\left(\frac{\log n}{\log \log n}\right)$, and none fell below the $k < \frac{\log n}{\log \log n}$ [2] bound for a thin rectangle. In Tile Automata without cooperative binding, is it possible to optimally construct $k \times n$ thin rectangles?
- We allow transition rules between non-bonded tiles. Can the same results be achieved with the restriction that a transition rule can only exist between two tiles if they share an affinity in the same direction?

- While we show optimal bounds can be achieved without cooperative binding, can we simulate so-called zig-zag aTAM systems? These are a restricted version of the cooperative aTAM that is capable of Turing computation.
- We show efficient bounds for constructing strings in Tile Automata. Given the power of the model, it should be possible to build algorithmically defined shapes such as in [33] by printing Komolgorov optimal strings and inputting them to a Turing machine.

References

- 1 Leonard Adleman, Qi Cheng, Ashish Goel, and Ming-Deh Huang. Running time and program size for self-assembled squares. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 740–748, 2001.
- 2 Gagan Aggarwal, Qi Cheng, Michael H Goldwasser, Ming-Yang Kao, Pablo Moisset De Espanes, and Robert T Schweller. Complexities for generalized models of self-assembly. *SIAM Journal on Computing*, 34(6):1493–1515, 2005.
- 3 John Calvin Alumbaugh, Joshua J. Daymude, Erik D. Demaine, Matthew J. Patitz, and Andréa W. Richa. Simulation of programmable matter systems using active tile-based self-assembly. In Chris Thachuk and Yan Liu, editors, *DNA Computing and Molecular Programming*, pages 140–158, Cham, 2019. Springer International Publishing.
- 4 Bahar Behsaz, Ján Maňuch, and Ladislav Stacho. Turing universality of step-wise and stage assembly at temperature 1. In Darko Stefanovic and Andrew Turberfield, editors, *DNA Computing and Molecular Programming*, pages 1–11, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- 5 David Caballero, Timothy Gomez, Robert Schweller, and Tim Wylie. Verification and Computation in Restricted Tile Automata. In Cody Geary and Matthew J. Patitz, editors, *26th International Conference on DNA Computing and Molecular Programming (DNA 26)*, volume 174 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:18, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DNA.2020.10.
- 6 Sarah Cannon, Erik D. Demaine, Martin L. Demaine, Sarah Eisenstat, Matthew J. Patitz, Robert T. Schweller, Scott M Summers, and Andrew Winslow. Two Hands Are Better Than One (up to constant factors): Self-Assembly In The 2HAM vs. aTAM. In *30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013)*, volume 20 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 172–184. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013.
- 7 Angel A Cantu, Austin Luchsinger, Robert Schweller, and Tim Wylie. Signal passing self-assembly simulates tile automata. In *31st International Symposium on Algorithms and Computation (ISAAC 2020)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.
- 8 Cameron Chalk, Austin Luchsinger, Eric Martinez, Robert Schweller, Andrew Winslow, and Tim Wylie. Freezing simulates non-freezing tile automata. In David Doty and Hendrik Dietz, editors, *DNA Computing and Molecular Programming*, pages 155–172, Cham, 2018. Springer International Publishing.
- 9 Cameron Chalk, Eric Martinez, Robert Schweller, Luis Vega, Andrew Winslow, and Tim Wylie. Optimal staged self-assembly of general shapes. *Algorithmica*, 80(4):1383–1409, 2018.
- 10 Gourab Chatterjee, Neil Dalchau, Richard A. Muscat, Andrew Phillips, and Georg Seelig. A spatially localized architecture for fast and modular DNA computing. *Nature Nanotechnology*, July 2017. URL: <https://www.microsoft.com/en-us/research/publication/spatially-localized-architecture-fast-modular-dna-computing/>.
- 11 Matthew Cook, Yunhui Fu, and Robert Schweller. Temperature 1 self-assembly: Deterministic assembly in 3d and probabilistic assembly in 2d. In *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms*, pages 570–589. SIAM, 2011.

- 12 Erik D Demaine, Martin L Demaine, Sándor P Fekete, Mashhood Ishaque, Eynat Rafalin, Robert T Schweller, and Diane L Souvaine. Staged self-assembly: nanomanufacture of arbitrary shapes with $o(1)$ glues. *Natural Computing*, 7(3):347–370, 2008.
- 13 Alberto Dennunzio, Enrico Formenti, Luca Manzoni, Giancarlo Mauri, and Antonio E Porreca. Computational complexity of finite asynchronous cellular automata. *Theoretical Computer Science*, 664:131–143, 2017.
- 14 David Doty, Jack H Lutz, Matthew J Patitz, Robert T Schweller, Scott M Summers, and Damien Woods. The tile assembly model is intrinsically universal. In *2012 IEEE 53rd Annual Symposium on Foundations of Computer Science*, pages 302–310. IEEE, 2012.
- 15 Nazim Fates. A guided tour of asynchronous cellular automata. In *International Workshop on Cellular Automata and Discrete Complex Systems*, pages 15–30. Springer, 2013.
- 16 Bin Fu, Matthew J Patitz, Robert T Schweller, and Robert Sheline. Self-assembly with geometric tiles. In *International Colloquium on Automata, Languages, and Programming*, pages 714–725. Springer, 2012.
- 17 David Furcy, Scott M. Summers, and Logan Withers. Improved Lower and Upper Bounds on the Tile Complexity of Uniquely Self-Assembling a Thin Rectangle Non-Cooperatively in 3D. In Matthew R. Lakin and Petr Šulc, editors, *27th International Conference on DNA Computing and Molecular Programming (DNA 27)*, volume 205 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DNA.27.4.
- 18 Oscar Gilbert, Jacob Hendricks, Matthew J Patitz, and Trent A Rogers. Computing in continuous space with self-assembling polygonal tiles. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 937–956. SIAM, 2016.
- 19 Eric Goles, P-E Meunier, Ivan Rapaport, and Guillaume Theyssier. Communication complexity and intrinsic universality in cellular automata. *Theoretical Computer Science*, 412(1-2):2–21, 2011.
- 20 Eric Goles, Nicolas Ollinger, and Guillaume Theyssier. Introducing freezing cellular automata. In *Cellular Automata and Discrete Complex Systems, 21st International Workshop (AUTOMATA 2015)*, volume 24, pages 65–73, 2015.
- 21 Leopold N Green, Hari KK Subramanian, Vahid Mardanlou, Jongmin Kim, Rizal F Hariadi, and Elisa Franco. Autonomous dynamic control of DNA nanostructure self-assembly. *Nature chemistry*, 11(6):510–520, 2019.
- 22 Daniel Hader and Matthew J Patitz. Geometric tiles and powers and limitations of geometric hindrance in self-assembly. *Natural Computing*, 20(2):243–258, 2021.
- 23 Jacob Hendricks, Matthew J. Patitz, Trent A. Rogers, and Scott M. Summers. The power of duples (in self-assembly): It’s not so hip to be square. *Theoretical Computer Science*, 743:148–166, 2018. doi:10.1016/j.tcs.2015.12.008.
- 24 Ming-Yang Kao and Robert Schweller. Reducing tile complexity for self-assembly through temperature programming. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, SODA '06, pages 571–580, USA, 2006. Society for Industrial and Applied Mathematics.
- 25 Pierre-Etienne Meunier, Matthew J. Patitz, Scott M. Summers, Guillaume Theyssier, Andrew Winslow, and Damien Woods. Intrinsic universality in tile self-assembly requires cooperation. In *Proceedings of the 2014 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 752–771, 2014. doi:10.1137/1.9781611973402.56.
- 26 Pierre-Étienne Meunier and Damien Regnault. Directed Non-Cooperative Tile Assembly Is Decidable. In Matthew R. Lakin and Petr Šulc, editors, *27th International Conference on DNA Computing and Molecular Programming (DNA 27)*, volume 205 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:21, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DNA.27.6.

- 27 Pierre-Étienne Meunier and Damien Woods. The non-cooperative tile assembly model is not intrinsically universal or capable of bounded Turing machine simulation. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2017, pages 328–341, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3055399.3055446.
- 28 Turlough Neary and Damien Woods. P-completeness of cellular automaton rule 110. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Automata, Languages and Programming*, pages 132–143, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- 29 Nicolas Ollinger and Guillaume Theyssier. Freezing, bounded-change and convergent cellular automata. *arXiv preprint*, 2019. arXiv:1908.06751.
- 30 Matthew J. Patitz, Robert T. Schweller, and Scott M. Summers. Exact shapes and Turing universality at temperature 1 with a single negative glue. In *Proceedings of the 17th International Conference on DNA Computing and Molecular Programming*, DNA’11, pages 175–189, Berlin, Heidelberg, 2011. Springer-Verlag.
- 31 Paul WK Rothmund and Erik Winfree. The program-size complexity of self-assembled squares. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 459–468, 2000.
- 32 Nicholas Schiefer and Erik Winfree. Time complexity of computation and construction in the chemical reaction network-controlled tile assembly model. In Yannick Rondelez and Damien Woods, editors, *DNA Computing and Molecular Programming*, pages 165–182, Cham, 2016. Springer International Publishing.
- 33 David Soloveichik and Erik Winfree. Complexity of self-assembled shapes. *SIAM Journal on Computing*, 36(6):1544–1569, 2007.
- 34 Anupama J. Thubagere, Wei Li, Robert F. Johnson, Zibo Chen, Shayan Doroudi, Yae Lim Lee, Gregory Izatt, Sarah Wittman, Niranjan Srinivas, Damien Woods, Erik Winfree, and Lulu Qian. A cargo-sorting DNA robot. *Science*, 357(6356):eaan6558, 2017. doi:10.1126/science.aan6558.
- 35 Erik Winfree. *Algorithmic Self-Assembly of DNA*. PhD thesis, California Institute of Technology, June 1998.
- 36 Damien Woods, Ho-Lin Chen, Scott Goodfriend, Nadine Dabby, Erik Winfree, and Peng Yin. Active self-assembly of algorithmic shapes and patterns in polylogarithmic time. In *Proceedings of the 4th conference on Innovations in Theoretical Computer Science*, pages 353–354, 2013.
- 37 Thomas Worsch. Towards intrinsically universal asynchronous ca. *Natural Computing*, 12(4):539–550, 2013.

Loosely-Stabilizing Phase Clocks and The Adaptive Majority Problem

Petra Berenbrink ✉

Universität Hamburg, Germany

Felix Biermeier ✉

Universität Hamburg, Germany

Christopher Hahn ✉

Universität Hamburg, Germany

Dominik Kaaser ✉ 

Universität Hamburg, Germany

Abstract

We present a loosely-stabilizing phase clock for population protocols. In the population model we are given a system of n identical agents which interact in a sequence of randomly chosen pairs. Our phase clock is leaderless and it requires $O(\log n)$ states. It runs forever and is, at any point of time, in a synchronous state w.h.p. When started in an arbitrary configuration, it recovers rapidly and enters a synchronous configuration within $O(n \log n)$ interactions w.h.p. Once the clock is synchronized, it stays in a synchronous configuration for at least $\text{poly}(n)$ parallel time w.h.p.

We use our clock to design a loosely-stabilizing protocol that solves the adaptive variant of the majority problem. We assume that the agents have either opinion A or B or they are undecided and agents can change their opinion at a rate of $1/n$. The goal is to keep track which of the two opinions is (momentarily) the majority. We show that if the majority has a support of at least $\Omega(\log n)$ agents and a sufficiently large bias is present, then the protocol converges to a correct output within $O(n \log n)$ interactions and stays in a correct configuration for $\text{poly}(n)$ interactions, w.h.p.

2012 ACM Subject Classification Theory of computation

Keywords and phrases Population Protocols, Phase Clocks, Loose Self-stabilization, Clock Synchronization, Majority, Adaptive

Digital Object Identifier 10.4230/LIPIcs.SAND.2022.7

Related Version *arXiv version*: <https://doi.org/10.48550/arXiv.2106.13002> [13]

1 Introduction

In this paper we introduce a loosely-stabilizing leaderless phase clock for the population model and demonstrate its usability by applying the clock to the comparison problem introduced in [3]. Population protocols have been introduced by Angluin et al. [5]. A population consists of n anonymous agents. A random scheduler selects in discrete time steps pairs of agents to interact. The interacting agents execute a state transition, as specified by the *algorithm* of the population protocol. Angluin et al. [5] gave a variety of motivating examples for the population model, including averaging in sensor networks, or modeling a disease monitoring system for a flock of birds. In [24] the authors introduce the notion of loose-stabilization. A population protocol is loosely-stabilizing if, from an arbitrary state, it reaches a state with correct output fast and remains in such a state for a polynomial number of interactions. In contrast, self-stabilizing protocols are required to converge to the correct output state from any possible initial configuration and stay in a correct configuration indefinitely. Many population protocols heavily rely on so-called *phase clocks* which divide the interactions into



© Petra Berenbrink, Felix Biermeier, Christopher Hahn, and Dominik Kaaser;
licensed under Creative Commons License CC-BY 4.0

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).

Editors: James Aspnes and Othon Michail; Article No. 7; pp. 7:1–7:17

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

blocks of $O(n \log n)$ interactions each. The phase clocks are used to synchronize population protocols. For example, in [22, 16] they are used to efficiently solve leader election and in [15] they are used to solve the majority problem.

In the first part of this paper we present a loosely-stabilizing and leaderless phase clock with $O(\log n)$ many states per agent. We show that this clock can run forever and that, at any point of time, it is synchronized w.h.p.¹ In contrast to related work [1, 7, 15, 22], our clock protocol *recovers* rapidly in case of an error: from an arbitrary configuration it always enters a synchronous configuration within $O(n \log n)$ interactions w.h.p. Once synchronized it stays in a synchronous configuration for at least $\text{poly}(n)$ interactions, w.h.p. Our phase clock can be used to synchronize population protocols into phases of $O(n \log n)$ interactions, guaranteeing that there is a big overlap between the phases of any pair of agents. Our clock protocol is simple, robust and easy to use.

In the second part of this paper we demonstrate how to apply our phase clock by solving an *adaptive majority* problem motivated by the work of [3, 4]. Our problem is defined as follows. Each agent has either opinion A , B , or U for being undecided. We say that agents change their input with rate r if in every time step an arbitrary agent can change its opinion with probability r . The goal is to output, at any time, the actual majority opinion. The idea of our approach is as follows. Our protocol simply starts, at the beginning of each phase, a *static* majority protocol as a black box. This protocol takes as an input the set of opinions at that time and calculates the majority opinion over these inputs. The outcome of the protocol is then used during the whole next phase as majority opinion. In order to highlight the simplicity of our phase clock, we first use the very natural protocol based solely on canceling opposing opinions introduced in [7]. Then we present a variant based on the undecided state dynamics from [8] which works as follows. The agents have one of two opinions A or B , or they are undecided. Whenever two agents with the same opinion interact, nothing happens. When two agents with an opposite opinion interact they will become undecided. Undecided agents interacting with an agent with either opinion A or opinion B adopt that opinion.

Without loss of generality we assume that A is the majority opinion in the following. When at least $\Omega(\log n)$ agents have opinion A , there is a constant factor bias between A and B , and the opinions change at most at rate $1/n$ per interaction, the system outputs A w.h.p. Our protocol requires only $O(\log n)$ many states. For the setting where all agents have either opinion A or B (none of the agent is in the undecided state U) and we have an additive bias of $n^{3/4+\varepsilon}$ for some constant $\varepsilon > 0$ is present, the system again converges to A w.h.p. In the latter setting we can tolerate a rate of order $r = \Omega(n^{-1/4+\varepsilon})$.

Related Work. Population protocols have been introduced by Angluin et al. [5]. Many of the early results focus on characterizing the class of problems which are solvable in the population model. For example, population protocols with a constant number of states can exactly compute predicates which are definable in Presburger arithmetic [5, 6, 9]. There are many results for majority and leader election, see [20] and [16] for the latest results. In [24] the authors introduce the notion of *loose-stabilization* to mitigate the fact that self-stabilizing protocols usually require some global knowledge on the population size (or a large amount of states). See [17] for an overview of self-stabilizing population protocols.

In [7] the authors present and analyze a phase clock which divides time into phases of $O(n \log n)$ interactions assuming that a unique leader exists. They also present a generalization using a junta of size n^ε (for constant ε) instead of a unique leader and analyze the

¹ The expression *with high probability* (w.h.p.) refers to a probability of $1 - n^{-\Omega(1)}$.

process empirically. In [22] the authors show that a junta can be elected using $O(\log \log n)$ many states and use the resulting clock to solve leader election. The protocol can easily be modified such that it requires only a constant number of states after the junta election [15]. In [1] the authors present a leaderless phase clock with $O(\log n)$ states. In contrast to our leaderless phase clock, the clock from [1] was not proven to be self-stabilizing. The analysis is based on the potential function analysis introduced in [25] for the greedy balls-into-bins strategy where each ball has to be allocated into one out of two randomly chosen bins. This analysis assumes an initially balanced configuration and it cannot be adopted to an arbitrary unbalanced state, which would be required to deal with unsynchronized clock configurations. In [10] the authors consider a variant of the population model, so-called clocked population protocols, where agents have an additional flag for clock ticks. The clock signal indicates when the agents have waited sufficiently long for a protocol to have converged. They show that a clocked population protocol running in less than ω^k time for fixed $k \geq 2$ is equivalent in power to nondeterministic Turing machines with logarithmic space.

Another line of related work considers the problem of *exact majority*, where one seeks to achieve (guaranteed) majority consensus, even if the additive bias is as small as one [21, 1, 14, 12]. The currently best protocol [20] solves exact majority with $O(\log n)$ states and $O(\log n)$ stabilization time, both in expectation and w.h.p. The authors of [8] solve the approximate majority problem. They introduce the undecided state dynamics in the population model for two opinions. They show that their 3-state protocol reaches consensus w.h.p. in $O(n \log n)$ interactions. If the bias is of order $\omega(\sqrt{n} \cdot \log n)$ the undecided state dynamics converges towards the initial majority w.h.p. In [18] the required bias is reduced to $\Omega(\sqrt{n \log n})$. For completeness [11] provides a survey about further protocols in the gossip model.

In [2] the authors define the **catalytic input model** (CI model). In this model the agents are divided into the two groups catalytic agents and non-catalytic agents. Non-catalytic agents perform pairwise interactions and change their state. Catalytic agents never change their state. Additionally to the normal state changes non-catalytic agents can perform spurious state changes; the so-called leak rate specifies the frequency of the spurious reactions. The goal of the non-catalytic agents is to compute a function over the states of the catalytic agents. The authors develop an algorithm for their model to detect whether there is a catalytic agent in a given state D or not. Note that, due to the leaky transactions non-catalysts can compute false-positives. In [4] the authors use the catalytic input model with n catalysts and m non-catalysts which they call worker agents ($N = n + m$). They solve the approximate majority problem for two opinions w.h.p. in $O(N \log N)$ interactions when the initial bias among the catalysts is $\Omega(\sqrt{N \log N})$ and $m = \Theta(n)$. They show that the size of the initial bias is tight up to a $O(\sqrt{\log N})$ factor. Additionally, they consider the approximate majority problem in the CI model and in the population model with leaks. Their protocols tolerate a leak rate of at most $\beta = O(\sqrt{N \log N}/N)$ in the CI model and a leak rate of at most $\beta = O(\sqrt{n \log n}/n)$ in the population model. They also show a separation between the computational power of the CI model and the population model.

In [3] the authors consider the CI model and introduce the *robust comparison problem*. The catalytic agents are either in state A or B and the goal of the worker agents is to decide which of the two states A and B has the larger support. In their dynamic version the number of agents in state A or B can change during the execution as long as the counts for A and B remain stable for a sufficiently long period allowing the algorithm to stabilize on an output. If at time t at least $\Omega(\log n)$ catalytic agents are in either A or B and the ratio between the numbers of agents supporting agents A and B is at least a constant, then most non-catalytic agents (up to $O(n/\log n)$ agents) outputs w.h.p. the correct majority. The protocol needs with $O(\log n \cdot \log \log n)$ states per agent, assuming that the number of catalytic agents in A and B does not change in the meantime. They also mention that with standard population

splitting $O(\log n + \log \log n)$ states are sufficient under the constraint that only a constant fraction of the agents store the output. Additionally the authors show that their protocol is robust to leaky transitions at a rate of $O(1/n)$. If the initial support of A and B states is $\Omega(\log^2 n)$ the authors can strengthen their results such that a ratio between the two base states of $1 + o(1)$ is sufficient.

2 Population Model and Problem Definitions

In the population model we are given a set V of n anonymous *agents*. At each time step two agents are chosen independently and uniformly at random randomly to *interact*. We assume that interactions between two agents (u, v) are ordered and call u the *initiator* and v the *responder*. The interacting agents update their states according to a common transition function of their previous states. Formally, a *population protocol* is defined as a tuple of a finite set of *states* Q , a *transition function* $\delta : Q \times Q \rightarrow Q \times Q$, a finite set of *output symbols* Σ , and an *output function* $\omega : Q \rightarrow \Sigma$ which maps every state to an output. A *configuration* is a mapping $C : V \rightarrow Q$ which specifies the state of each agent. An execution of a protocol is an infinite sequence C_0, C_1, \dots such that for all C_i there exist two agents v_1, v_2 and a transition $(q_1, q_2) \rightarrow (q'_1, q'_2)$ such that $C_i(v_1) = q_1, C_i(v_2) = q_2, C_{i+1}(v_1) = q'_1, C_{i+1}(v_2) = q'_2$ and $C_i(w) = C_{i+1}(w)$ for all $w \neq v_1, v_2$. The main quality criteria of a population protocol are the required number of states and the running time measured in interactions.

The goal of this paper is to develop protocols that are *loosely-stabilizing* according to the definitions of [24]. Let \mathcal{C} denote an arbitrary subset of all possible configurations. Consider an infinite sequence of configurations C_0, C_1, \dots . For an arbitrary configuration $C_i \notin \mathcal{C}$ the *convergence time* is defined as the smallest t such that $C_{i+t_1} \in \mathcal{C}$. Intuitively, the convergence time bounds the time it takes to reach a configuration in \mathcal{C} when starting from a configuration not in \mathcal{C} . For an arbitrary configuration $C_i \in \mathcal{C}$ the *holding time* t_2 is defined as the largest t such that $C_{i+t_2} \in \mathcal{C}$. Intuitively, the holding time bounds the time during which the protocol remains in a configuration in \mathcal{C} when starting from a configuration in \mathcal{C} .

► **Definition 1.** A protocol is loosely-stabilizing wrt. to a subset of configurations \mathcal{C} if the maximum convergence time over all possible configurations is w.h.p. less than t_1 and the minimum holding time over all configurations in \mathcal{C} is w.h.p. at least t_2 .

Phase Clocks. Phase clocks are used to synchronize population protocols. We assume a phase clock is implemented by simple counters $\text{clock}[u_1], \dots, \text{clock}[u_n]$ modulo $|Q|$ (see, e.g., [1, 7, 15, 19, 22]). Whenever $\text{clock}[u]$ crosses zero, agent u receives a so-called *signal*. These signals will divide the time into *phases* of $\Theta(n \log n)$ interactions each. We say a (τ, w) -phase clock is synchronous in the time interval $[t_1, t_2]$ if every agent gets a signal every $\Theta(n \log n)$ interactions. More formally:

- Every agent receives a signal in the first $2 \cdot (w + 1) \cdot \tau \cdot n$ steps of the interval.
- Assume an agent u receives a signal at time $t \in [t_1, t_2]$.
 - For all $v \in V$, agent v receives a signal at time t_v with $|t - t_v| \leq \tau \cdot n$.
 - Agent u receives the next signal at time t' with $(w + 1) \cdot \tau \cdot n \leq |t - t'| \leq 2 \cdot (w + 1) \cdot \tau \cdot n$.

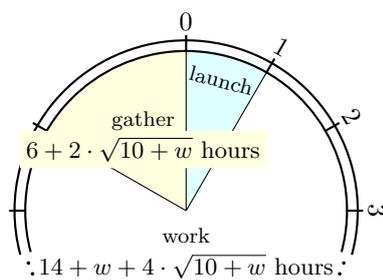
The above definition divides the time interval $[t_1, t_2]$ into a sequence of subintervals that alternates between so-called burst-intervals and overlap-intervals.

- A burst-interval has length at most $\tau \cdot n$ and every agent gets exactly one signal.
- An overlap-interval consists of those time steps between two burst-intervals where none of the agents gets a signal. It has length at least $w \cdot \tau \cdot n$.

A burst-interval together with the subsequent overlap-interval forms a *phase*.

To define loosely-stabilizing phase clocks, we need to define the set of *synchronous* configurations \mathcal{C} . Intuitively, we call a state C_t of a (τ, w) -phase clock at time t synchronous if the counters of all pairs of agents do not deviate much. More precisely, $\text{clock}[u](t) - \text{clock}[v](t) <_{|Q|} f(w, \tau)$ for all pairs of agents (u, v) (Here, “ $\leq_{|Q|}$ ” denotes smaller w.r.t. the circular order modulo $|Q|$.) We define f and give the formal definition of a synchronous configuration in the next section.

3 Clock Algorithm



■ **Figure 1** Schematic representation of the clock states.

In this section we introduce our phase clock protocol. For ease of notation, we assume in this section that the state space of an agent is Q . Our (τ, w) -phase clock has a state space $Q = \{0, \dots, (21 + w + 6 \cdot \sqrt{10 + w}) \cdot \tau - 1\}$. The clock states are divided into $(21 + w + 6 \cdot \sqrt{10 + w})$ hours, and each hour consists of $\tau = \tau(c) = 36 \cdot (c + 4) \cdot \ln n$ minutes. The parameter $c \geq 6$ determines the error probability in each phase and thus the holding time (see Theorem 2). The parameter $w \geq 0$ can be chosen as needed by the application. As we will see, τ is a multiple of the running time of the *one-way epidemic* (see Lemma 4) and $w \cdot \tau \cdot n$ is the number of interactions in which our agents are synchronized. We divide the hours into three consecutive intervals (see Figure 1): the launching interval I_{launch} (first hour), the working interval I_{work} ($14 + w + 4 \cdot \sqrt{10 + w}$ hours) and the gathering interval I_{gather} (last $6 + 2 \cdot \sqrt{10 + w}$ hours). We say that agent u is in one of the intervals whenever its clock counter $\text{clock}[u]$ is in that interval. If the agents are either all in I_{gather} , all in I_{work} , or all in I_{launch} , we say the configuration is *homogeneous*. For two agents u and v we define a *distance* $d(u, v) = \min\{|\text{clock}[u] - \text{clock}[v]|, |Q| - |\text{clock}[u] - \text{clock}[v]|\}$ that takes the cyclic nature of the clock into account. This allows us to define synchronous configurations as follows.

► **Definition** (Synchronous Configuration). A configuration C is called *synchronous* if and only if for all pairs of agents (u, v) we have $d(u, v) < |I_{\text{launch}}| + |I_{\text{gather}}| = (7 + 2 \cdot \sqrt{10 + w}) \cdot \tau$.

Our clock works as follows. Assume agents (u, v) interact. With two exceptions, agent u increments its counter $\text{clock}[u]$ by one minute modulo $|Q|$ (Rules 1 and 2). If, however, u is in I_{gather} and v is in I_{launch} then agent u adopts $\text{clock}[v]$ (Rule 3): we say the agent *hops*. If u is in I_{gather} and v is in I_{work} then agent u returns to the beginning of I_{gather} (Rule 4): we say that the agent *resets*. We define that agent u receives a *signal* whenever its clock crosses the wrap-around from I_{gather} to I_{launch} . Formally, our clock uses the following transitions.

$$(q_1, q_2) \in (Q \setminus I_{\text{gather}}) \times Q: \quad (q_1, q_2) \rightarrow (q_1 + 1, q_2) \quad (\text{step forward}) \quad (1)$$

$$(q_1, q_2) \in I_{\text{gather}} \times I_{\text{gather}}: \quad (q_1, q_2) \rightarrow (q_1 + 1 \bmod |Q|, q_2) \quad (\text{step forward}) \quad (2)$$

$$(q_1, q_2) \in I_{\text{gather}} \times I_{\text{launch}}: \quad (q_1, q_2) \rightarrow (q_2, q_2) \quad (\text{hopping}) \quad (3)$$

$$(q_1, q_2) \in I_{\text{gather}} \times I_{\text{work}}: \quad (q_1, q_2) \rightarrow (|I_{\text{launch}}| + |I_{\text{work}}|, q_2) \quad (\text{reset}) \quad (4)$$

Note that $|I_{\text{work}}| = 2(|I_{\text{launch}}| + |I_{\text{gather}}|) + w \cdot \tau$ to have $w \cdot \tau \cdot n$ homogeneous working configurations between two signals. On the other hand, $|I_{\text{gather}}|/\tau = \Theta(\sqrt{|I_{\text{work}}|/\tau})$ which is necessary to apply Chernoff bounds. We chose $|I_{\text{launch}}| = \tau$ for simplicity. On an intuitive level, the clock works as follows. Assume the clock is synchronized and all agents are in I_{launch} . Now consider the next $k = \Theta(n \cdot |Q|)$ interactions. All agents step forward according to Rule 1 until they reach I_{gather} . The maximum distance between any agents grows during the k interactions but can still be bounded by $O(\sqrt{k/n}) = O(\sqrt{|w| \cdot \log n})$, w.h.p. via Chernoff bounds. Hence, due to the choice of w there is no agent left behind in I_{launch} when the first agent reaches I_{gather} . Additionally, due to the size of I_{gather} when the last agent enters I_{gather} all of the other agents are still in $|I_{\text{gather}}|$. As soon as the first agent reaches I_{launch} , Rule 3 (agents hop onto agents in I_{launch}) ensures that all agents start the next phase without a large gap. Hence, there is an interaction after which all agents are in $|I_{\text{launch}}|$ which brings us back to our initial configuration (all agents in I_{launch}).

Now we consider an asynchronous configuration C_t where the agents can be arbitrarily distributed over the $|Q|$ states of the clock. The main idea for the recovery of our clock is as follows. We show that after $O(n \log n)$ interactions there is a time t where I_{launch} is empty. After $O(n \log n)$ additional steps most of the agents are in I_{gather} : agents cannot hop since I_{launch} is empty, and they reset as soon as they interact with an agent in I_{work} . They enter I_{launch} as soon as the first agent crosses 0 by increasing its clock counter.

We will show that the following two properties hold for our clock.

► **Theorem 2.** *Let $\tau = 36 \cdot (c + 4) \cdot \ln n$ and let w be a sufficiently large constant. Let t_1, t_2 with $t_1 \leq t_2$ be two points in time and assume that the configuration C_{t_1} at time t_1 is a homogeneous launching configuration and $t_2 - t_1 \leq n^c$. Then the clock counters of the agents implement a synchronous (τ, w) -phase clock in the time interval $[t_1, t_2]$ w.h.p.*

► **Theorem 3.** *The clock counters of the agents implement a $(O(n \cdot \log n), \Omega(\text{poly } n))$ -loosely-stabilizing $(\Theta(\log n), w)$ -phase clock.*

Note that our simulations suggest that the algorithm also works if τ is smaller by a constant fraction. We prove Theorem 2 in Section 4 and Theorem 3 in Section 5.

Auxiliary Results. The one-way-epidemic is a population protocol with state space $\{0, 1\}$ and transitions $(q_1, q_2) \rightarrow (\max(q_1, q_2), q_2)$. An agent in state 0 is called *susceptible* and an agent in state 1 is called *infected*. We say agent v infects agent u if v is infected and u initiates an interaction with v . The following result is folklore, see, e.g., [7]. Additional details can be found in the full version of this paper.

► **Lemma 4 (One-way-epidemic).** *Assume an agent starts the one-way epidemic in step 1. All agents are infected after $t = \tau/4 \cdot n$ many steps with probability at least $1 - n^{-(7+2c)}$.*

The following lemma bounds the number of interactions initiated by some fixed agent u among a sequence of t interactions. It is used throughout Sections 4 and 5 and follows immediately from Chernoff bounds (see [23]).

► **Lemma 5.** *Consider an arbitrary sequence of t interactions and let X_u be the number of interactions initiated by agent u within this sequence. Then*

$$\Pr[X_u < (1 + \delta) \cdot t/n] \geq 1 - n^{-\frac{12 \cdot (c+4)t \cdot \delta^2}{n \cdot \tau}} \quad \text{and} \quad \Pr[X_u > (1 - \delta) \cdot t/n] \geq 1 - n^{-\frac{18 \cdot (c+4)t \cdot \delta^2}{n \cdot \tau}}.$$

4 Maintenance: Proof of Theorem 2

In this section we first show the following main result. At the end of the section we show how Theorem 2 follows from this proposition.

► **Proposition 6 (Maintenance).** *Consider our (τ, w) -phase clock for n agents with $\tau = 36 \cdot (c+4) \cdot \ln n$ for any $c \geq 6$ and sufficiently large w . Let configuration C_{t_1} be a homogeneous launching configuration. Then, with probability at least $1 - n^{-(c+1)}$, there exists a $t_2 = \Theta(n \cdot w \cdot \log n)$ such that the following holds:*

1. $C_{t_1+t_2}$ is a homogeneous launching configuration,
2. $\forall t \in [t_1, t_1 + t_2]: C_t$ is synchronous,
3. in the time interval $[t_1, t_1 + t_2]$ there exists a contiguous sequence of homogeneous working configurations of length $w \cdot \tau \cdot n$.

We split the proof of Proposition 6 into two parts, Lemmas 7 and 8. The formal proof follows.

Proof. Assume the configuration C_{t_1} at time t_1 is a homogeneous launching configuration. Statements 1 and 2 of Proposition 6 follow immediately from Lemmas 7 and 8:

- It follows from Lemma 7 that the agents transition via a sequence of synchronous configurations into a homogeneous gathering configuration within $\Theta(n \cdot w \cdot \log n)$ time w.h.p.
- It follows from Lemma 8 that the agents transition via a sequence of synchronous configurations back into a homogeneous launching configuration within $\Theta(n \cdot w \cdot \log n)$ further time w.h.p.

It remains to show Statement 3. Recall that in a synchronous configuration all pairs of agents have distance (w.r.t. the circular order modulo $|Q|$) at most $\Delta = (7 + 2 \cdot \sqrt{10 + w}) \cdot \tau$. Since $|I_{\text{work}}| = w \cdot \tau + 2\Delta$ it immediately follows that there must be $w \cdot \tau \cdot n$ interactions where all agents are in I_{work} . This concludes the proof. ◀

The following lemma establishes that w.h.p. all agents transition from a homogeneous launching configuration into a homogeneous gathering configuration via a sequence of synchronous configurations.

► **Lemma 7.** *Let C_t be a homogeneous launching configuration. Let $t' = n \cdot \frac{|I_{\text{launch}}| + |I_{\text{work}}|}{1 - (2 \cdot \sqrt{|I_{\text{work}}|/\tau})^{-1}}$. Then the following holds with probability at least $1 - n^{-(c+1)}/2$:*

1. $C_{t+t'}$ is a homogeneous gathering configuration and
2. $\forall t'' \in [t, t + t'] : C_{t''}$ is synchronous.

Proof. In the following we assume w.l.o.g. $t = 0$. We prove the two statements separately.

Statement 1. Our goal is to show that after t' interactions all agents are in I_{gather} when we start from a homogeneous launching configuration C_0 at time $t = 0$. We first show that there is no agent left in I_{launch} when the first agent enters I_{gather} . Let t_a be the first interaction in which an agent enters I_{gather} . Note that before t_a all agents are either in I_{launch} or in I_{work} and thus the agents increase their counter by one whenever they initiate an interaction.

First we show that w.h.p. $t_a \geq 2 \cdot \tau \cdot n$. Let $X_u(2 \cdot \tau \cdot n)$ denote the number of interactions agent u initiates before time $2 \cdot \tau \cdot n$. From Lemma 5 it follows with $\delta = 1$ that $X_u(2 \cdot \tau \cdot n) < 4 \cdot \tau$ with probability at least $1 - n^{-24 \cdot (c+4)}$. Since $4 \cdot \tau < |I_{\text{work}}|$, it holds that

$\text{clock}[u](2 \cdot \tau \cdot n) < |I_{\text{launch}}| + |I_{\text{work}}|$ in this case. Hence, agent u has not yet reached I_{gather} with probability at least $1 - n^{-24 \cdot (c+4)}$ at time t_a . It follows from a union bound over all agents that no agent has reached I_{gather} with probability at least $1 - n^{-24 \cdot (c+4)+1}$ at time t_a .

Next we show that w.h.p. at time $2 \cdot \tau \cdot n$ all agents have left I_{launch} . As before, let $X_u(2 \cdot \tau \cdot n)$ denote the number of interactions agent u initiates before time $2 \cdot \tau \cdot n$. From Lemma 5 it follows with $\delta = 1$ that $X_u(2 \cdot \tau \cdot n) > \tau$ with probability at least $1 - n^{-36 \cdot (c+4)}$. Since $\tau = |I_{\text{launch}}|$, it holds that $\text{clock}[u](t + 2 \cdot \tau \cdot n) \geq |I_{\text{launch}}|$ in this case. Hence, agent u has left I_{launch} with probability at least $1 - n^{-36 \cdot (c+4)}$ at time t_a . Again, it follows from a union bound over all agents that all agents have left I_{launch} with probability at least $1 - n^{-36 \cdot (c+4)+1}$ at time t_a .

Let now t_b be the first interaction in which an agent enters the last minute of I_{gather} and observe that $t_b > t_a$. Then, w.h.p. no agent is in I_{launch} during the time interval $[t + t_a, t + t_b]$. Therefore, agents cannot hop. Thus, by definition of t_b , no agent can leave I_{gather} before time t_b . All initiators must therefore either increase their counter by one or reset.

First we show that w.h.p. $t_b > t'$. From Lemma 5 it follows with $\delta = \left(2 \cdot \sqrt{|I_{\text{work}}|/\tau}\right)^{-1}$ that $X_u(t') < |I_{\text{work}}| + |I_{\text{gather}}|$ with probability at least $1 - n^{-3(c+4)}$. (Note that we use $(1 + \delta)/(1 - \delta) < 1/(1 - 2 \cdot \delta)$ for $\delta < 0.5$ and $(|I_{\text{launch}}| + |I_{\text{work}}|)/(1 - 2 \cdot \delta) = |I_{\text{work}}| + (w + 5 \cdot \sqrt{10 + w + 16})/(\sqrt{10 + w + 1}) \cdot \tau < |I_{\text{work}}| + |I_{\text{gather}}|$.) Thus, $\text{clock}[u](t') \leq \text{clock}[u](0) + X_u(t') < |I_{\text{launch}}| - 1 + |I_{\text{work}}| + |I_{\text{gather}}|$ (which is the last state of I_{gather}) with probability at least $1 - n^{-3(c+4)}$. By a union bound, this holds for all agents with probability at least $1 - n^{-(3c+11)}$.

Next we show that w.h.p. at time t' all agents have reached I_{gather} . From Lemma 5 it follows for our choice of δ that $X_u(t') > |I_{\text{launch}}| + |I_{\text{work}}|$ with probability at least $1 - n^{-9 \cdot (c+4)/2}$. Thus, $\text{clock}[u](t') \geq \text{clock}[u](0) + X_u(t') > 0 + |I_{\text{launch}}| + |I_{\text{work}}|$, with probability at least $1 - n^{-9 \cdot (c+4)/2}$. By a union bound, this holds for all agents with probability at least $1 - n^{-(17+9/2 \cdot c)}$.

Together it follows that at time t' no agent has left I_{gather} but all agents have entered it with probability at least $1 - n^{-(c+3)}$. Therefore, $C_{t'}$ is a homogeneous gathering configuration.

Statement 2. Recall that a synchronous configuration C is defined as a configuration where $\max_{(u,v)} \{d(u,v)\} < |I_{\text{launch}}| + |I_{\text{gather}}|$. As before, let $X_u(i)$ denote the number of interactions agent u initiates before time i . Now fix a time $t \leq t'$ and a pair of agents (u, v) with $X_u(t) < X_v(t)$. We use Lemma 5 to bound the deviation of $X_u(t)$ and $X_v(t)$ at time t as follows: $\Pr[X_u(t) > t/n - |I_{\text{gather}}|/2] \geq 1 - n^{-6(c+4)}$ and $\Pr[X_v(t) < t/n + |I_{\text{gather}}|/2] \geq 1 - n^{-4(c+4)}$. Therefore, $|X_v(t) - X_u(t)| < |I_{\text{gather}}|$ with probability at least $1 - n^{-4(c+4)} - n^{-6(c+4)}$.

Note that Lemma 5 allows us to bound the deviation in the numbers of interactions initiated by agents u and v . However, this does not immediately give a bound on the difference of the clock counters $|\text{clock}[v](t) - \text{clock}[u](t)|$. To bound the deviation of clock counters (by $|I_{\text{launch}}| + |I_{\text{gather}}|$), we therefore distinguish three cases.

First, assume that neither u nor v have reached I_{gather} at time t . Then $\text{clock}[u](t) = \text{clock}[u](0) + X_u(t)$ and $\text{clock}[v](t) = \text{clock}[v](0) + X_v(t)$. Observe that by the assumption of the lemma, both u and v are in I_{launch} at time $t = 0$ and thus $|\text{clock}[v](0) - \text{clock}[u](0)| < |I_{\text{launch}}|$. Together with the above bound on $|X_v(t) - X_u(t)|$ we get $|\text{clock}[v](t) - \text{clock}[u](t)| < |I_{\text{launch}}| + |I_{\text{gather}}|$.

Secondly, assume that u has not reached I_{gather} but v has reached I_{gather} at time t . Then $\text{clock}[u](t) = \text{clock}[u](0) + X_u(t)$. For $\text{clock}[v](t)$, however, it might have occurred that v has reset in some interactions before time t . Nevertheless, the clock counter of v is bounded by

the number of initiated interactions such that $\text{clock}[v](t) \leq \text{clock}[v](t) + X_v(t)$. (Note that v can only increment its $\text{clock}[v]$ counter or reset its value; hopping is not possible since we have shown in the proof of the first statement that I_{launch} is empty when the first agent enters I_{gather} .) Therefore, we get again $|\text{clock}[v](t) - \text{clock}[u](t)| < |I_{\text{launch}}| + |I_{\text{gather}}|$.

Finally, assume that both u and v are in I_{gather} at time t . Then $|\text{clock}[v](t) - \text{clock}[u](t)| \leq |I_{\text{gather}}| < |I_{\text{launch}}| + |I_{\text{gather}}|$ is trivially true.

There are no further cases: in the proof of the first statement we have shown that all agents transition from a homogeneous launching configuration to a homogeneous gathering configuration during the time interval $[0, t']$. The result now follows from a union bound over all $o(n^2)$ points in time $t \leq t'$ and all $n \cdot (n - 1)$ pairs of agents. ◀

The following lemma is the main technical contribution of this section. It establishes that w.h.p. all agents transition from a homogeneous gathering configuration into a homogeneous launching configuration via a sequence of synchronous configurations. Consider a homogeneous gathering configuration and recall that whenever an agent hops from I_{gather} into I_{launch} it adopts the state of the responder. The main difficulty is to show that all agents hop into I_{launch} before the first agent leaves I_{launch} .

► **Lemma 8.** *Let C_t be a homogeneous gathering configuration. Then with probability at least $1 - n^{-(c+1)}/2$ the following holds:*

1. *there exists a $t_0 = O(n \cdot \sqrt{w} \cdot \log n)$ such that the first agent enters I_{launch} at time $t + t_0$,*
2. *there exists a $t' \leq \tau/4 \cdot n$ such that $C_{t+t_0+t'}$ is a homogeneous launching configuration,*
3. *$\forall t'' \in [t, t + t'] : C_{t''}$ is synchronous.*

Proof. We prove first show Statement 1, Statement 2 and Statement 3 are shown together.

Statement 1. Let t_0 be defined such that the first agent u leaves I_{gather} at time $t + t_0$. Since C_t is a homogeneous gathering configuration, I_{launch} is empty at time t and hence agent u can only leave I_{gather} by increasing its counter. In every interaction before time $t + t_0$ some agent has to increase its state by one. Thus $t_0 \leq n \cdot |I_{\text{gather}}| = O(n \cdot \sqrt{w} \cdot \log n)$.

Statement 2+3. We continue our analysis at time t_0 and again assume w.l.o.g. for the sake of brevity of notation that $t_0 = 0$. Note that at that time exactly one agent is in state 0 and all remaining agents are still in I_{gather} . We show the following: there exists a time $\tilde{t} = \tau/4 \cdot n$ such that at time \tilde{t} all agents are in I_{launch} (Recall that $|I_{\text{launch}}| = \tau \cdot n$). To do so we first define a simplified process with the same state space Q , however, we refer to the last state of I_{launch} as **stop**. Agents in **stop** never change their state (which renders the states of I_{work} unreachable). The formal definition of the simplified process is as follows. Rule 2 and 3 are identical to the original process and Rule 1 and 4 are modified as follows.

$$(q_1, q_2) \in (I_{\text{launch}} \setminus \{\text{stop}\}) \times Q: \quad (q_1, q_2) \rightarrow (q_1 + 1, q_2) \quad (\text{step forward}) \quad (1)$$

$$(q_1, q_2) \in \{\text{stop}\} \times Q: \quad (q_1, q_2) \rightarrow (q_1, q_2) \quad (\text{stopping}) \quad (4)$$

For this simplified process we show a lower bound: after $\tilde{t} = \Theta(n \cdot \log n)$ interactions all agents are in I_{launch} . Then we show (for the simplified process) an upper bound: in $C_{\tilde{t}}$ none of the agents are in state **stop**. A simple coupling of the simplified process and the original process shows that under these circumstances none of the agents entered I_{work} for our original process. This finishes the proof with $t' = \tilde{t}$.

Lower Bound. In the simplified process agents can enter I_{launch} either via hopping or by making enough steps forward on their own. From Lemma 4 it follows that all agents enter I_{launch} after at most $\tilde{t} = \tau/4 \cdot n$ interactions with probability at least $1 - n^{-(5+c)}$. (For

the upper bound, one can simply discard setting the clock counter to zero when an agent enters I_{launch} by increasing its counter.) Showing that none of the agents are in state **stop** is much harder. Due to the hopping the clock counters of agents in I_{launch} are highly correlated. Nevertheless, we can show that the clock counters of each agent can be majorized by independent binomially distributed random variables as follows.

Upper Bound. Let u_i be the i 'th agent that enters I_{launch} and let t_i be the time when u_i enters I_{launch} . Let furthermore $X_i(t)$ be a random variable for the clock counter of agent u_i in I_{launch} in the time interval $[0, t]$. Formally, we define for a time step t that $X_i(t) = 0$ if u_i is in I_{gather} and $X_i(t) = \text{clock}[u_i](t)$ if u_i is in I_{launch} . We show by induction on i that $X_i(t)$ is majorized by a random variable $Z_i(t)$ with binomial distribution $Z_i(t) \sim \text{Bin}(t, 1/n \cdot (1 + 1/(n-1))^{i-1})$, i.e., $\Pr[X_i(t) > x] \leq \Pr[Z_i(t) > x]$ for all $x \geq 0$. Ultimately, our goal is to apply Chernoff bounds to $Z_i(\tilde{t})$ which shows that agent u_i does not reach **stop** w.h.p. The statement for the simplified process then follows from a union bound over all agents w.h.p.

Base Case. For the base case we consider all agents that enter I_{launch} on their own by incrementing their counters to 0 (modulo $|Q|$) in I_{gather} . Fix such an agent u_i . It holds that $X_i(t)$ for $t \geq t_i$ has binomial distribution $X_i(t) \sim \text{Bin}(t - t_i, 1/n)$. Therefore, $X_i(t) \prec Z_i(t)$ as claimed.² (Intuitively, this means that the clock counter of any other agent u_i with $i > 1$ that enters I_{launch} at time $t_i > 0$ is majorized by the clock counter of an agent which enters I_{launch} at time $t_1 = 0$ and increments its counter with probability $1/n$.)

Induction Step. For the induction step we now consider all agents that enter I_{launch} by hopping onto some other agent in I_{launch} . Fix such an agent u_i . Let S_i be the event that agent u_i is the i 'th agent that enters I_{launch} . Let furthermore t_i be the time when u_i enters I_{launch} . We condition on S_i and observe that agent u_i enters I_{launch} by hopping onto some other agent $u_j \in \{u_1, \dots, u_{i-1}\}$. Intuitively, we would now like to exploit the fact that the counter of agent u_i is copied at time t_i from agent u_j such that $X_i(t_i) = X_j(t_i)$. Unfortunately, we must be extremely careful here: conditioning on S_i alters the probability space! (For example, under S_i the agent u_i with $i \geq 3$ cannot initiate an interaction with agent u_1 before agent u_2 does, since S_i rules out that u_i enters I_{launch} before agent u_2 .) We account for the modified probability space as follows.

Let $\Omega_{S_i}(t)$ be the probability space of possible interactions conditioned on S_i at time $t \leq \tilde{t}$. Without the conditioning on S_i , the probability space $\Omega(t)$ at time t contains all (ordered) pairs of agents with $|\Omega(t)| = n \cdot (n-1)$. When conditioning on S_i , the event S_i rules out that agent u_i interacts with any other agent $u_j \in I_{\text{launch}}$ before time t_i . In particular, agent u_i cannot interact with another agent u_j with $j < i$ during the time interval $[t_j, t_i]$. In order to give a lower bound on $|\Omega_{S_i}(t)|$, we exclude all $(n-1)$ interactions (u_i, u_j) for $j \in [n]$ from $\Omega(t)$. Hence $|\Omega_{S_i}(t)| \geq n \cdot (n-1) - (n-1) = (n-1)^2$ for any time $t \leq t_i$. (The probability space after time t_i is not affected by conditioning on S_i , but the majorization holds nonetheless.) We now consider the event $E_{\hat{t}}$ for $\hat{t} \leq t_i$ that the interaction at time \hat{t} increments $X_j(t)$ by 1 (recall that u_j is the agent onto which u_i hopped). It then holds for the reduced probability space Ω_{S_i} that $\Pr[E_{\hat{t}} | S_i] \leq \Pr[E_{\hat{t}}] \cdot |\Omega(t)| / |\Omega_{S_i}(t)|$. (Note that Ω_{S_i} is still a uniform probability space.) We calculate

$$\frac{|\Omega(t)|}{|\Omega_{S_i}(t)|} = \frac{n \cdot (n-1)}{(n-1)^2} = 1 + \frac{1}{n-1}$$

² The expression $X \prec Y$ means that the random variable X is majorized by the random variable Y .

and get $\Pr[E_{\hat{t}} | S_i] \leq \Pr[E_{\hat{t}}] \cdot (1 + 1/(n-1))$ for $\hat{t} \leq t_i$. Therefore, we use the induction hypothesis (that describes $X_j(t_i)$) and get $X_i(t_i) \prec Z_i(t_i)$, where $Z_i(t_i) \sim \text{Bin}(t_i, 1/n \cdot (1 + 1/(n-1))^{i-1})$. Similarly, we define $E_{\hat{t}}$ for $\hat{t} \geq t_i$ to be the event that u_i increments its counter in I_{launch} . Observe that $\Pr[E_{\hat{t}}] \leq 1/n$ for $\hat{t} > t_i$. It follows that $X_i(t) \prec Z_i(t)$ with distribution $Z_i(t) \sim \text{Bin}(\hat{t}, 1/n \cdot (1 + 1/(n-1))^{i-1})$ for $t \leq \hat{t}$ as claimed. This concludes the induction.

Conclusions. From the induction it follows that for each agent u_i the clock counter $\text{clock}[u_i](\tilde{t})$ at time \tilde{t} is majorized by a random variable $Z(\tilde{t})$ with binomial distribution $Z(\tilde{t}) \sim \text{Bin}(\tilde{t}, e/n)$. (Note that we used the inequality $(1 + 1/(n-1))^{(n-1)} < e$.) From Chernoff bounds (see [23]) it follows that $\Pr[Z(\tilde{t}) \geq \tau - 1] \leq n^{-(c+4)}$. Finally, the proof for the simplified process follows from a union bound over all agents.

It is now straightforward to couple the actual phase clock process with the simplified process. Assume that we start both processes at time 0 when exactly one agent is in state 0. In the simplified process no agent reaches state τ in $\tau/4 \cdot n$ interactions with probability at least $1 - n^{-(c+3)}$. In this case, however, the simplified process and the actual phase clock process do not deviate and, in particular, no agent reaches the beginning of I_{work} in $\tau/4 \cdot n$ many interactions. Thus, the configuration $C_{t'}$ is a homogeneous launching configuration with probability at least $1 - n^{-(c+3)}$.

Since all agents started in I_{gather} and no agent reaches the beginning of I_{work} , the agents are in a synchronous configuration by definition during the whole time interval $[0, t']$. ◀

We are now ready to put everything together and prove our first theorem.

Proof of Theorem 2. The proof of Theorem 2 follows readily from the main result of this section, Proposition 6.

Assume the configuration at time t_1 is a homogeneous launching configuration. Then from Proposition 6 it follows w.h.p. that after $t_2 = \Theta(n \cdot w \cdot \log n)$ interactions the configuration C_{t_2} is again a homogeneous launching configuration, and all configurations in $[t_1, t_2]$ are synchronous. From Statement 3 it follows that no agent receives a signal in a contiguous subinterval $[t'_1, t'_2] \subset [t_1, t_2]$ of length $t'_2 - t'_1 = w \cdot \tau \cdot n$. This shows that we have w.h.p. the required *overlap* according to the definition of synchronous (τ, w) -phase clocks.

From Lemma 8 it follows w.h.p. that all agents transition from a homogeneous gathering configuration into a homogeneous launching configuration within $\tau/4 \cdot n$ interactions. Recall that whenever an agent crosses zero, it receives a signal. Therefore, when all agents transition from a homogeneous gathering configuration into a homogeneous launching configuration via a sequence of synchronous configurations, all agents receive exactly one signal, and the time between two signals of two agents (u, v) is w.h.p. at most $\tau/4 \cdot n$. This shows that we have w.h.p. the required *bursts* according to the definition of synchronous (τ, w) -phase clocks.

Together, the counters of our clock implement a synchronous (τ, w) -phase clock in $[t_1, t_2]$ with probability n^{-c} . It follows from an inductive argument that the clock counters implement a synchronous (τ, w) -phase clock during the n^c interactions that follow time t_1 w.h.p. ◀

5 Recovery: Proof of Theorem 3

In this section we first show the following main result. At the end of the section we show how Theorem 3 follows from this proposition.

► **Proposition 9** (Recovery). *Consider our (τ, w) -phase clock with n agents and sufficiently large c and w . Let C_{t_1} be an arbitrary configuration. Then with probability at least $1 - 1/n$, there exists a $t_2 = O(n \cdot w \cdot \log n)$ such that $C_{t_1+t_2}$ is a homogeneous launching configuration.*

We say a configuration is an *almost homogeneous gathering configuration* if no agent is in I_{launch} and at least $0.9 \cdot n$ many agents are in I_{gather} . We start our analysis by showing that within $t = O(n \cdot w \cdot \log n)$ interactions, we reach an almost homogeneous gathering configuration C_{t_1+t} .

► **Lemma 10.** *Let C_t be an arbitrary configuration. Then with probability at least $1 - 1/(3n)$, there exists a $t' = O(n \cdot w \cdot \log n)$ such that $C_{t+t'}$ is an almost homogeneous gathering configuration.*

Proof Sketch. The main idea of the proof is as follows. If there are not too many agents in I_{gather} , the reset rule prevents agents from reaching the end of I_{gather} . Agents may still enter I_{launch} by hopping, but if no agent enters state 0, eventually there is no agent left in state 0 to hop on. Then the same argument applies to state 1, and so on. Eventually, there are no agents left in I_{launch} to hop onto. This means the agents are *trapped* in I_{gather} until a sufficiently large number of agents enters I_{gather} which renders resetting quite unlikely again. The resulting configuration is what we call an almost homogeneous gathering configuration. ◀

Next, we show that from an almost homogeneous gathering configuration we reach a homogeneous gathering configuration in $O(n \cdot w \cdot \log n)$ interactions. From Lemma 8 in Section 4 it then follows that we reach a homogeneous launching configuration in an additional number of $O(n \cdot \log n)$ interactions.

► **Lemma 11.** *Let C_t be an almost homogeneous gathering configuration. Then with probability at least $1 - 1/(3n)$, there exists a $t' = \Theta(n \cdot w \cdot \ln n)$ such that $C_{t+t'}$ is a homogeneous gathering configuration.*

Proof Sketch. If C_t is an almost homogeneous gathering configuration, then there are no agents in I_{launch} and at least $0.9 \cdot n$ many agents in I_{gather} . Thus, agents cannot hop until an agent enters I_{launch} on its own. Now there are two cases. If no agent enters I_{launch} on its own before the last agent enters I_{gather} , we are done: this is by definition of a homogeneous gathering configuration. Otherwise, we will show that a large fraction of agents leave I_{gather} together. This large fraction behaves similar as in the proof of the maintenance. The remaining agents have a small *head start* but then they are again *trapped* in I_{gather} until the bulk of agents arrives. Once the bulk of agents enters I_{gather} we have reached a homogeneous gathering configuration and all agents start to run through the clock synchronously. ◀

Proof of Theorem 3. The proof of Theorem 3 follows readily from the main result of this section, Proposition 9. Observe that $\tau = \Theta(\log n)$. According to Proposition 9, our clock recovers to a homogeneous launching configuration in $O(n \cdot \log n)$ interactions. By Theorem 2, this marks the beginning of a time interval in which the agents implement a synchronous (τ, w) -phase clock. It follows immediately from Theorem 2 that this interval has length n^c . Together, this implies that our (τ, w) -phase clock is a $(O(n \cdot \log n), \Omega(\text{poly}(n)))$ -loosely-stabilizing $(\Theta(\log n), w)$ -phase clock. ◀

6 Adaptive Majority Problem

In this section we consider the adaptive majority problem. At any time, every agent has as input either an *opinion* (A or B) or it has no input, in which case we say it is *undecided* (U). During the execution of the protocol, the opinions of the agents can change. In the

adaptive majority problem, the goal is that all agents output (at all times) the opinion which is dominant among all inputs. In this setting we present a *loosely-stabilizing* protocol that solves the adaptive majority problem. We define a loosely-stabilizing adaptive majority protocol according to Definition 1 by defining \mathcal{C} as all configurations where all agents output the correct majority opinion. Recall that the performance of a loosely-stabilizing protocol is measured in terms of the *convergence time* and the *holding time*. Note that the loose-stabilization comes from an application of our phase clock. The phase clocks guarantee synchronized phases for polynomial time. During this time we say a configuration C is *correct* w.r.t. the adaptive majority problem if the following conditions hold. Suppose there is a sufficiently large bias towards one opinion. Then every agent in a correct configuration outputs the majority opinion. Otherwise, if there is no sufficiently large bias, we consider any output of the agents as correct. In this setting, we show the following result: We show that a $(O(\log n), \text{poly}(n))$ -loosely-stabilizing algorithm exists that solves adaptive majority, using $O(\log n)$ states per agent.

6.1 Our Protocol

Our protocol is based on the (τ, w) -phase clock defined in Section 3 with $w = 566$. In addition to the states required by the clock, every agent v has three variables $\text{input}[v]$, $\text{opinion}[v]$, and $\text{output}[v]$. The variable $\text{input}[v]$ always reflects the current input to the agent, $\text{opinion}[v]$ holds the current opinion of agent v , and $\text{output}[v]$ defines the current output value of agent v . All three variables take values in $\{A, B, U\}$. A and B stand for the corresponding opinions and U stands for *undecided*. The state space of the protocol is $Q_c \times \{A, B, U\}^3$ where Q_c is the state space of our clock for $\tau = 36 \cdot (c + 4) \ln n$ and $w = 566$.

We use the (τ, w) -phase clock to synchronize the agents. Then it follows from Proposition 6 that all configurations are synchronous w.h.p. Observe that in a synchronous configuration for our choice of parameters the clock counters of agents do not deviate by more than $\Delta = 55 \cdot \tau$. This allows us to define three *subphases* of I_{work} , where agents execute three different protocols, as follows. We split the working interval I_{work} into six contiguous subintervals of equal length. The clock counters $\text{clock}[u]$ allows us to define a simple interface to the phase clock for each agent u as follows. The variable $\text{subphase}[u]$ for each agent u is then defined as follows. We set $\text{subphase}[u] = 1$ if $\text{clock}[u]$ is in the first subinterval of I_{work} , $\text{subphase}[u] = 2$ if $\text{clock}[u]$ is in the third subinterval of I_{work} , and $\text{subphase}[u] = 3$ if $\text{clock}[u]$ is in the fifth subinterval of I_{work} . Otherwise, $\text{subphase}[u] = \perp$. The clock now assures a clean separation into these subphases such that no two agents perform a different protocol at any time w.h.p. Additionally, we will show the overlap within each subphase is long enough such that the subprotocols for the corresponding subphases succeed w.h.p.

On an intuitive level, our protocol works as follows. At the beginning of the phase, the input is copied to the opinion variable. In the first protocol, the support of opinions A and B is amplified until no undecided agents are left. We call this the Pólya Subphase. In the second protocol, agents with opposite opinions cancel each other out, becoming undecided. We call this the Cancellation Subphase. Finally, in the third protocol the single remaining opinion is amplified again. We call this the Broadcasting Subphase. The resulting opinion is copied to the output variable after the working interval I_{work} . Formally, our protocol is specified in Algorithm 1.

In the remainder of this section, we let A_t and B_t denote the number of agents u with $\text{opinion}[u] = A$ and $\text{opinion}[u] = B$, respectively, at time t . Analogously, we let A_t^{IN} and B_t^{IN} denote the number of agents u with $\text{input}[u] = A$ and $\text{input}[u] = B$, respectively, at time t . We now state our main result for this section.

■ **Algorithm 1** Interaction of agents (u, v) in the adaptive majority protocol.

```

1 update clock[u] according to Rules 1–4 with  $w = 566$ 
2 if Agent  $u$  receives a signal then opinion[u]  $\leftarrow$  input[u]
3 if subphase[u] = 1  $\wedge$  opinion[u] =  $U$  then opinion[u]  $\leftarrow$  opinion[v]
4 if subphase[u] = 2  $\wedge$  opinion[u]  $\neq$  opinion[v]  $\wedge$  opinion[u], opinion[v]  $\neq U$  then
5   opinion[u], opinion[v]  $\leftarrow U$ 
6 if subphase[u] = 3  $\wedge$  opinion[u] =  $U$  then opinion[u]  $\leftarrow$  opinion[v]
7 if clock[u]  $\geq |I_{\text{launch}}| + |I_{\text{work}}|$  then output[u]  $\leftarrow$  opinion[u]

```

► **Theorem 12.** *Algorithm 1 is a $(O(n \log n), \Omega(\text{poly}(n)))$ -loosely stabilizing adaptive majority protocol.*

Note that we match the results of [3] for $r = 1/n$, a multiplicative bias of $1 + \beta = 1 + 1/\log n$ and $\Omega(\log^2 n)$ many agents ($\alpha \geq \log n$). In contrast to their protocol, every agent outputs at any point of time the correct majority opinion, w.h.p. But, again in contrast to their work, we do not consider leaky transitions.

6.2 Analysis

In the following analysis, we consider an arbitrary but fixed phase. We condition on the event that the clock is synchronized according to Proposition 6. We show the following main result, and later in this section we describe how Theorem 12 follows from it this proposition. The proofs for the statements in this section can be found in the full version of this paper.

► **Proposition 13.** *Assume that at time t_1 the agents are in a homogeneous launching configuration and we have $A_{t_1} \geq \alpha \cdot \log n$ and $A_{t_1} \geq (1 + \beta) \cdot B_{t_1}$. If α and β are large enough constants, then there exists a $t_2 = \Theta(n \cdot w \cdot \log n)$ such that all agents output A in configuration $C_{t_1+t_2}$ with probability $1 - n^{-c}$.*

The analysis is split into three parts, one for the *Pólya Subphase*, one for the *Cancellation Subphase*, and one for the *Broadcasting Subphase*. First, we assume that no changes in the input occur. Then we generalize our results: we adopt the undecided state dynamics introduced in [8], and show how we can tolerate input changes at various rates.

Observe that we get a separation between the subphases from the guarantees of the phase clock in Theorem 2: no two agents are more than $1/6$ of I_{work} apart. We also know that every agent has copied its input at the beginning of the phase before the first agent enters the first subphase. The total time for the three subphases (including the separation time) is sufficiently large such that every agent has finished its work before the next phase starts.

When we refer to a distribution *before a subphase*, we mean the distribution at the time just before the first agent performs an interaction in that subphase. Analogously, when we refer to a distribution *after a subphase*, we mean the distribution at the time when the last agent has performed an interaction in that subphase. Recall that in the following analysis, we let A_t and B_t denote the number of agents u with $\text{opinion}[u] = A$ and $\text{opinion}[u] = B$, respectively, at time t . Furthermore, we let s_i and e_i (for *start* and *end*) be the first and the last time, respectively, when an agent performs an interaction in the i 'th subphase.

Subphases. We first consider the Pólya Subphase, where we model the process by means of so-called Pólya urns. Pólya urns are defined as follows. Initially, the urn contains a red balls and b blue balls. In each step, a ball is drawn uniformly at random from the urn. The

ball's color is observed, and it is returned into the urn along with an additional ball of the same color. The Pólya-Eggenberger distribution $PE(a, b, m)$ describes the total number of red balls after m steps of this urn process.

This observation allows us to apply concentration bounds to the opinion distribution after the Pólya Subphase. Recall that s_1 and e_1 are the first and the last time steps, respectively, when an agent performs an interaction in the Pólya Subphase. We get the following lemma.

► **Lemma 14.** *Let $a = A_{s_1-1}$ and $b = B_{s_1-1}$. For any constant $\beta > 0$ there exists a constant α such that if $a > \alpha \cdot \log n$ and $a > (1 + \beta) \cdot b$ then $A_{e_1} - B_{e_1} = \Omega(n)$ with probability at least $1 - n^{-(c+2)}$.*

Next we consider the Cancellation Subphase. The goal is to remove any occurrence of the minority opinion. Whenever an agent with opinion A interacts with another agent with opinion B , both agents become undecided. Formally, we show the following lemma.

► **Lemma 15.** *If $A_{s_2-1} - B_{s_2-1} = \Omega(n)$ then $A_{e_2} = \Omega(n)$ and $B_{e_2} = 0$ with probability at least $1 - n^{-(c+2)}$.*

Finally we consider the Broadcasting Subphase. The goal is to spread the (unique) remaining opinion to all other agents. Whenever an undecided agent u interacts with another agent v that has an opinion, agent u adopts the opinion of agent v . This leads to a configuration where every agent has the majority opinion w.h.p. Formally, we show the following lemma.

► **Lemma 16.** *If $A_{e_2} = \Omega(n)$ and $B_{e_2} = 0$, then $A_{e_3} = n$ and $B_{e_3} = 0$ with probability at least $1 - n^{-(c+2)}$.*

We have now everything we need to prove Proposition 13 and in turn Theorem 12.

Proof of Proposition 13. We assume the configuration at time t_1 is a homogeneous launching configuration. From Proposition 6 it follows that all configurations in the time interval $[t_1, t_1 + t_2]$ for some $t_2 = \Theta(n \cdot w \cdot \log n)$ are synchronous with probability at least $1 - n^{-(c+1)}$. This means that the three subphases are strictly separated as explained above. It therefore follows (each with probability at least $1 - n^{-(c+2)}$), from Lemma 14 that after the Pólya Subphase no agent is undecided, from Lemma 15 that after the Cancellation Subphase no agent has opinion B , and from Lemma 16 that after the Broadcasting Subphase all agents have opinion A . Once all agents have opinion A , this becomes the output when the agents enter I_{gather} . Together, this shows that all agents output the majority opinion after $\Theta(n \cdot w \cdot \log n)$ interactions with probability at least $1 - n^{-c}$. ◀

Proof of Theorem 12. Here we show the result without input changes. Fix a time t_1 and assume the agents are in an arbitrary configuration at time t_1 . From Theorem 3 it follows the agents enter a synchronous configuration within $O(n \log n)$ interactions and stay in synchronous configurations for $\text{poly}(n)$ time w.h.p.

Now we consider a fixed synchronized phase $i < \text{poly}(n)$ of our phase clock. It follows from Proposition 13 that all agents enter a correct configuration at the end of phase i with probability at least $1 - n^{-c}$. (Recall that in a correct configuration all agents have to output the majority opinion if there is a sufficiently large bias. Without a bias, any output constitutes a correct configuration.) From the guarantees of the phase clock it follows that the first synchronized phase starts within $O(n \log n)$ time after time t_1 w.h.p. This shows a convergence time of $O(n \log n)$. From a union bound over at most n^{c-1} phases it follows that the protocol is in a correct configuration for $\text{poly}(n)$ interactions w.h.p. This shows a holding time of $\text{poly}(n)$.

The proof with input changes can be found in the full version. The main idea is that we bound the number of input changes in $\Theta(n \log n)$ interactions by a simple application of Chernoff bounds. ◀

Improving the Bound. In order to show-case the simplicity of the application of our phase clock, we have presented a simplistic protocol, where we assumed a constant factor bias towards the majority opinion. If we replace the Cancellation Subphase and the Broadcasting Subphase (lines 6 to 9 in Algorithm 1) with the *undecided state dynamics* introduced in [8] we can show a tighter result.

Formally, we show the following statement, the proof can be found in the full version of this paper.

► **Observation 17.** If we use the undecided state dynamics, Proposition 13 also holds for $\alpha = \Omega(\beta^{-2})$ provided that $\beta = \Omega(n^{-1/4+\varepsilon})$.

This means that we can solve the adaptive majority problem with a multiplicative bias of $1 + \beta = 1 + 1/\log n = 1 + o(1)$ and asymptotically at least $\Omega(\log^2 n)$ many agents with opinion A or B (assuming sufficiently large constants). Hence we achieve similar results as in [3] for a model without leaky transitions.

References

- 1 Dan Alistarh, James Aspnes, and Rati Gelashvili. Space-optimal majority in population protocols. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 2221–2239. SIAM, 2018. doi:10.1137/1.9781611975031.144.
- 2 Dan Alistarh, Bartłomiej Dudek, Adrian Kosowski, David Soloveichik, and Przemyslaw Uznanski. Robust detection in leak-prone population protocols. In *DNA Computing and Molecular Programming - 23rd International Conference, DNA*, volume 10467 of *Lecture Notes in Computer Science*, pages 155–171. Springer, 2017. doi:10.1007/978-3-319-66799-7_11.
- 3 Dan Alistarh, Martin Töpfer, and Przemyslaw Uznanski. Comparison dynamics in population protocols. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 55–65. ACM, 2021. doi:10.1145/3465084.3467915.
- 4 Talley Amir, James Aspnes, and John Lazarsfeld. Approximate majority with catalytic inputs. In Quentin Bramas, Rotem Oshman, and Paolo Romano, editors, *24th International Conference on Principles of Distributed Systems, OPODIS 2020, December 14-16, 2020, Strasbourg, France (Virtual Conference)*, volume 184 of *LIPICs*, pages 19:1–19:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.OPODIS.2020.19.
- 5 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Comput.*, 18(4):235–253, 2006. doi:10.1007/s00446-005-0138-3.
- 6 Dana Angluin, James Aspnes, and David Eisenstat. Stably computable predicates are semilinear. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing, PODC*, pages 292–299. ACM, 2006. doi:10.1145/1146381.1146425.
- 7 Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. *Distributed Comput.*, 21(3):183–199, 2008. doi:10.1007/s00446-008-0067-z.
- 8 Dana Angluin, James Aspnes, and David Eisenstat. A simple population protocol for fast robust approximate majority. *Distributed Comput.*, 21(2):87–102, 2008. doi:10.1007/s00446-008-0059-z.
- 9 Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Comput.*, 20(4):279–304, 2007. doi:10.1007/s00446-007-0040-2.

- 10 James Aspnes. Clocked population protocols. *J. Comput. Syst. Sci.*, 121:34–48, 2021. doi:10.1016/j.jcss.2021.05.001.
- 11 Luca Becchetti, Andrea E. F. Clementi, and Emanuele Natale. Consensus dynamics: An overview. *SIGACT News*, 51(1):58–104, 2020. doi:10.1145/3388392.3388403.
- 12 Stav Ben-Nun, Tsvi Kopelowitz, Matan Kraus, and Ely Porat. An $o(\log^{3/2} n)$ parallel time population protocol for majority with $o(\log n)$ states. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2020, Virtual Event, Italy, August 3-7, 2020*, page to appear, 2020.
- 13 Petra Berenbrink, Felix Biermeier, Christopher Hahn, and Dominik Kaaser. Self-stabilizing phase clocks and the adaptive majority problem. *CoRR*, abs/2106.13002, 2021. arXiv:2106.13002.
- 14 Petra Berenbrink, Robert Elsässer, Tom Friedetzky, Dominik Kaaser, Peter Kling, and Tomasz Radzik. Time-space trade-offs in population protocols for the majority problem. *Distributed Computing*, 2020.
- 15 Petra Berenbrink, Robert Elsässer, Tom Friedetzky, Dominik Kaaser, Peter Kling, and Tomasz Radzik. Time-space trade-offs in population protocols for the majority problem. *Distributed Comput.*, 34(2):91–111, 2021. doi:10.1007/s00446-020-00385-0.
- 16 Petra Berenbrink, George Giakkoupis, and Peter Kling. Optimal time and space leader election in population protocols. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC*, pages 119–129. ACM, 2020. doi:10.1145/3357713.3384312.
- 17 Janna Burman, Ho-Lin Chen, Hsueh-Ping Chen, David Doty, Thomas Nowak, Eric E. Severson, and Chuan Xu. Time-optimal self-stabilizing leader election in population protocols. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 33–44. ACM, 2021. doi:10.1145/3465084.3467898.
- 18 Anne Condon, Monir Hajiaghayi, David G. Kirkpatrick, and Ján Manuch. Approximate majority analyses using tri-molecular chemical reaction networks. *Nat. Comput.*, 19(1):249–270, 2020. doi:10.1007/s11047-019-09756-4.
- 19 Shlomi Dolev and Jennifer L. Welch. Self-stabilizing clock synchronization in the presence of byzantine faults. *J. ACM*, 51(5):780–799, September 2004. doi:10.1145/1017460.1017463.
- 20 David Doty, Mahsa Eftekhari, Leszek Gąsieniec, Eric Severson, Grzegorz Stachowiak, and Przemysław Uznański. A time and space optimal stable population protocol solving exact majority, 2022. FOCS 2022, to appear. arXiv:2106.10201.
- 21 David Doty and David Soloveichik. Stable leader election in population protocols requires linear time. *Distributed Computing*, 31(4):257–271, 2018. doi:10.1007/s00446-016-0281-z.
- 22 Leszek Gąsieniec and Grzegorz Stachowiak. Fast space optimal leader election in population protocols. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 2653–2667. SIAM, 2018. doi:10.1137/1.9781611975031.169.
- 23 Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005. doi:10.1017/CB09780511813603.
- 24 Yuichi Sudo, Junya Nakamura, Yukiko Yamauchi, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Loosely-stabilizing leader election in a population protocol model. *Theor. Comput. Sci.*, 444:100–112, 2012. doi:10.1016/j.tcs.2012.01.007.
- 25 Kunal Talwar and Udi Wieder. Balanced allocations: A simple proof for the heavily loaded case. In *Automata, Languages, and Programming - 41st International Colloquium, ICALP*, volume 8572 of *Lecture Notes in Computer Science*, pages 979–990. Springer, 2014. doi:10.1007/978-3-662-43948-7_81.

Complexity of Verification in Self-Assembly with Prebuilt Assemblies

David Caballero ✉

Department of Computer Science, University of Texas Rio Grande Valley, TX, USA

Timothy Gomez ✉

Department of Computer Science, University of Texas Rio Grande Valley, TX, USA

Robert Schweller ✉

Department of Computer Science, University of Texas Rio Grande Valley, TX, USA

Tim Wylie ✉

Department of Computer Science, University of Texas Rio Grande Valley, TX, USA

Abstract

We analyze the complexity of two fundamental verification problems within a generalization of the two-handed tile self-assembly model (2HAM) where initial system assemblies are not restricted to be singleton tiles, but may be larger pre-built assemblies. Within this model we consider the *producibility* problem, which asks if a given tile system builds, or produces, a given assembly, and the *unique assembly verification* (UAV) problem, which asks if a given system *uniquely* produces a given assembly. We show that producibility is NP-complete and UAV is coNP^{NP} -complete even when the initial assembly size and temperature threshold are both bounded by a constant. This is in stark contrast to results in the standard model with singleton input tiles where producibility is in P and UAV is in coNP for $\mathcal{O}(1)$ bounded temperature and coNP-complete when temperature is part of the input. We further provide preliminary results for producibility and UAV in the case of 1-dimensional *linear* assemblies with pre-built assemblies, and provide polynomial time solutions.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness; Applied computing → Computational biology

Keywords and phrases 2-handed assembly, verification, prebuilt

Digital Object Identifier 10.4230/LIPIcs.SAND.2022.8

Funding This research was supported in part by National Science Foundation Grant CCF-1817602.

1 Introduction

Self-Assembly is the process by which a system of simple particles autonomously come together to form complex structures. *Algorithmic* self-assembly studies scenarios in which dynamics of system molecules encode computation, allowing for algorithmic control of the self-assembly of matter. A premiere model for the study of algorithmic self-assembly is the *tile self-assembly model* [3, 17], in which system monomers are modeled as four-sided Wang tiles that randomly collide and combine based on matching tile edges and a given bonding threshold called the *temperature*. Tile self-assembly has received substantial theoretical consideration (see [13, 14, 19] for surveys and recent results) as well as various experimental DNA implementations [6, 10, 20].

In this paper we focus on a specific generalization of the standard 2-handed tile self-assembly model (2HAM) in which we permit initial assemblies to consist of *prebuilt* assemblies of more than one tile. The motivation for studying such a generalization is strong. First, some of the most successful implementations of algorithmic DNA self-assembly utilize a combination of singleton DNA tiles mixed with larger prebuilt assemblies. For example, the experimental implementations of DNA tile counters [10] and the 21 DNA tile circuits



© David Caballero, Timothy Gomez, Robert Schweller, and Tim Wylie; licensed under Creative Commons License CC-BY 4.0

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).

Editors: James Aspnes and Othon Michail; Article No. 8; pp. 8:1–8:15

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

implemented in [20] both utilize a combination of single tiles seeded with a larger prebuilt DNA origami structure encoding the program input. In [10], they additionally include a mix of singleton and prebuilt domino assemblies among the system's tile set. Second, the inclusion of prebuilt shapes of different geometries and sizes allows for the potential application of *steric hindrance*, in which geometric blocking of potential attachments is used to control algorithmic growth, as seen in the theoretical works of [5, 11, 12], and the experimental works of [9, 18]. These examples suggest that consideration of shapes more general than uniform single squares has the promise to allow for improved computational power and efficiency of self-assembled systems.

Given the importance of understanding self-assembly with prebuilt initial assemblies, we consider the complexity of two fundamental computational questions related to verifying the correctness of such systems. First is the *Producibility* problem, which asks if a given tile system can build/produce a given assembly. The second is the *Unique Assembly Verification* (UAV) problem, which asks if a given tile system *uniquely* produces a given assembly, i.e., produces the assembly and nothing else provided sufficient assembly time. For the producibility problem, the 2HAM with just single-tile initial assemblies has a polynomial time solution [7], whereas we show NP-completeness when prebuilt assemblies are permitted. In the case of the UAV problem with singleton tile initial assemblies, the problem resides in coNP [3] for a constant-bounded temperature threshold and is coNP-complete for larger temperature thresholds [15], whereas we show coNP^{NP}-completeness with prebuilt assemblies. In both scenarios, our hardness results hold even for prebuilt assemblies of a bounded $\mathcal{O}(1)$ size and $\mathcal{O}(1)$ -bounded temperature thresholds. We accompany these results with a preliminary exploration of the producibility and UAV problems when restricted to 1-dimensional *linear* assemblies with pre-built assemblies, and provide polynomial time solutions.

1.1 Previous work

The model used here differs from the polyTAM model of [11] in that the set of starting elements in our system are defined as assemblies made up of multiple tiles. In the polyTAM, the set of elements are single tiles that are allowed to be larger than a unit square. We are attempting to model the situations where smaller components may have preassembled into larger structures prior to being introduced to the system. This is more similar to the staged model of self-assembly however in that model all the bins are usually assumed to have the same temperature so any assemblies built in early stages must be producible. Here we only require that the input assemblies are stable.

Verification problems have been well-studied in many models of Tile Self-Assembly. In the Abstract Tile Assembly model (aTAM), both the producibility and UAV problem are solvable in polynomial time [1]. When allowing negative or repulsive glues the UAV problem becomes undecidable due to detachment [8], but when restricted to growth-only systems (no detachment can occur) the problem is coNP-complete [4]. Producibility verification in the 2-handed assembly model at any temperature, along with UAV for temperature 1, are both solvable in polynomial time [7]. Membership in the class coNP for general 2HAM systems was shown in [3] along with a hardness result showing coNP-completeness when one step into the 3rd dimension is allowed. By allowing the temperature to be a part of the input UAV has been shown to be coNP-complete [15] even in 2 dimensions. More powerful generalizations have shown an increase in complexity of the UAV problem such as in Tile Automata which merges ideas from Cellular Automata and the 2HAM. This problem was shown to be coNP^{NP}-complete even with the restrictions of Freezing (A tile only changes states a finite number of times) and growth only (no detachment).

■ **Table 1** Complexity of verifying producibility of an assembly in various models. The Assembly Size column indicates the size of the assemblies in the initial assembly set. Previous work has studied the case when only single tiles are allowed. Our results allow for up to constant sized assemblies.

Model	Input Assembly Size	Temperature	Result	Reference
aTAM	Single Tiles	Variable	P	[1]
2HAM	Single Tiles	Variable	P	[7]
2HAM	Constant	2	NP-complete	Thm. 3

■ **Table 2** Complexity results of Unique Assembly Verification in the aTAM and the 2HAM. UAV is undecidable in the negative aTAM, but coNP-complete with negative glues if the system never allows detachment (*growth only). **Tile Automata with freezing and growth only restrictions.

Model	Input Assembly	Temperature	Result	Reference
aTAM	Single Tiles	Variable	P	[1]
Neg. aTAM G.O.*	Single Tiles	2	coNP-complete	[4]
2HAM	Single Tiles	1	P	[7]
2HAM	Single Tiles	Constant	coNP	[3]
2HAM	Single Tiles	Variable	coNP-complete	[15]
2HAM 3D	Single Tiles	2	coNP-complete	[3]
Tile Automata**	Single Tiles	2	coNP ^{NP} -complete	[2]
2HAM	Constant	2	coNP ^{NP} -complete	Thm. 6

2 Definitions

In this section we overview the basic definitions related to the two-handed self-assembly model and the verification problems under consideration.

Tiles. A *tile* is a non-rotatable unit square with each edge labeled with a *glue* from a set Σ . Each pair of glues $g_1, g_2 \in \Sigma$ has a non-negative integer *strength* $\text{str}(g_1, g_2)$.

Configurations, bond graphs, and stability. A *configuration* is a partial function $A : \mathbb{Z}^2 \rightarrow T$ for some set of tiles T , i.e. an arrangement of tiles on a square grid. For a given configuration A , define the *bond graph* G_A to be the weighted grid graph in which each element of $\text{dom}(A)$ is a vertex, and the weight of the edge between a pair of tiles is equal to the strength of the coincident glue pair. A configuration is said to be τ -*stable* for positive integer τ if every edge cut of G_A has strength at least τ , and is τ -*unstable* otherwise.

Assemblies. For a configuration A and vector $\vec{u} = \langle u_x, u_y \rangle$ with $u_x, u_y \in \mathbb{Z}^2$, $A + \vec{u}$ denotes the configuration $A \circ f$, where $f(x, y) = (x + u_x, y + u_y)$. For two configurations A and B , B is a *translation* of A , written $B \simeq A$, provided that $B = A + \vec{u}$ for some vector \vec{u} . For a configuration A , the *assembly* of A is the set $\tilde{A} = \{B : B \simeq A\}$. An assembly \tilde{A} is a *subassembly* of an assembly \tilde{B} , denoted $\tilde{A} \sqsubseteq \tilde{B}$, provided that there exists an $A \in \tilde{A}$ and $B \in \tilde{B}$ such that $A \sqsubseteq B$. An assembly is τ -*stable* provided the configurations it contains are τ -stable. Assemblies \tilde{A} and \tilde{B} are τ -*combinable* into an assembly \tilde{C} provided there exist $A \in \tilde{A}$, $B \in \tilde{B}$, and $C \in \tilde{C}$ such that $A \cup B = C$, $A \cap B = \emptyset$, and \tilde{C} is τ -stable.

Two-handed assembly. A Two-handed assembly system is an ordered tuple (S, τ) where S is a set of *initial* assemblies and τ is a positive integer parameter called the *temperature*. Each assembly in S must be τ -stable. For a system (S, τ) , the set of *producible* assemblies $P'_{(S, \tau)}$ is defined recursively as follows:

1. $S \subseteq P'_{(S, \tau)}$.
2. If $A, B \in P'_{(S, \tau)}$ are τ -combinable into C , then $C \in P'_{(S, \tau)}$.

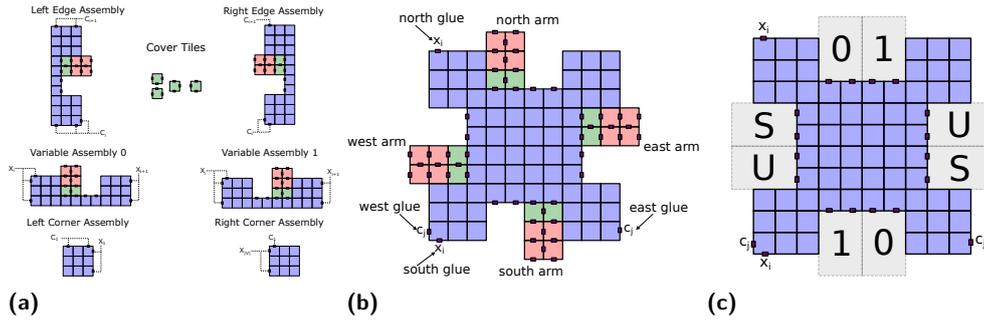


Figure 1 (a) Edge Assemblies used to construct the frame of the assembly. For each clause we include a left and right edge assembly. For each variable we include two variable assemblies representing 0 and 1. We include a single left corner assembly along with a right corner assembly. The filler tiles are used to fill in holes between attached macroblocks. (b) A single macro block $m_{i,j}(0, U, U)$ with outer glues labeled. The north and south glues connect to other macro blocks that represent the same variable. The east and west glues connects to other macroblocks that represent the same clause. (c) Arm position labels on a macroblock. Opposite sides have complementary values to allow for attachment.

A producible assembly is *terminal* provided it is not τ -combinable with any other producible assembly, and the set of all terminal assemblies of a system (S, τ) is denoted $P_{(S, \tau)}$. Intuitively, $P'_{(S, \tau)}$ represents the set of all possible assemblies that can self-assemble from the initial set S , whereas $P_{(S, \tau)}$ represents only the set of assemblies that cannot grow any further. An assembly A is *uniquely produced* if $P_{(S, \tau)} = \{A\}$ and for each $B \in P'_{(S, \tau)}$ $B \sqsubseteq A$.

► **Definition 1** (Producibility Problem). *Given a 2HAM system $\Gamma = (S, \tau)$ and an assembly A , is A a producible assembly of Γ ?*

► **Definition 2** (Unique Assembly Verification Problem (UAV)). *Given a 2HAM system $\Gamma = (S, \tau)$ and an assembly A , is A uniquely produced by Γ ?*

3 Producibility Hardness

In this section we show that the producibility problem is NP-complete if the initial set of assemblies may include assemblies larger than singleton tiles. The hardness is derived by reducing from 3SAT and holds even if assembly size and system temperature is bounded by a constant. Our construction extends to the seeded *abstract tile assembly model* where there is a seed tile, and elements of the tile set (in this case assembly set) attach one at a time to the growing seed. See appendix for details.

3.1 Overview

We reduce from 3SAT, which asks whether a given 3CNF formula ϕ is satisfiable. Let $|V|$ and $|C|$ be the number of variables and clauses in ϕ , respectively. The reduction creates an instance of producibility (Γ, A) with $\tau = 2$, such that Γ produces the target assembly A iff ϕ is satisfiable. The target assembly is a rectangle shown and described in Figure 4a. The assemblies in the reduction can be divided into two groups: edge assemblies (Figure 1a) and macroblocks (Figure 1b). There are two variable assemblies for each bit that each correspond to an assignment of 0 or 1 based on the position of the 3×2 *arm* section of the assembly

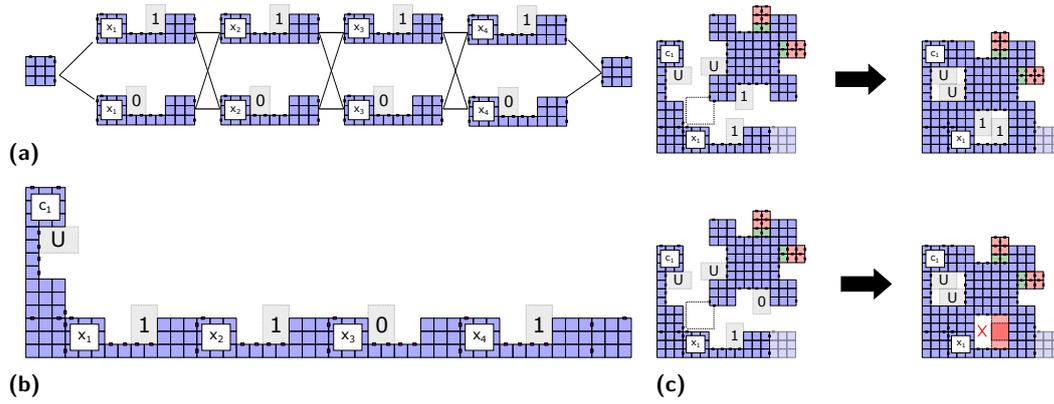


Figure 2 (a) Variable gadgets can non-deterministically build a frame assembly for each possible assignment to ϕ . (b) Example of the assembly made for assignment 1101 with a single left edge assembly attached. (c) If the macroblock has complimentary arm positions it will be able to attach to the frame assembly. If the macroblocks do not have matching arm positions the attachment will be geometrically blocked and cannot occur.

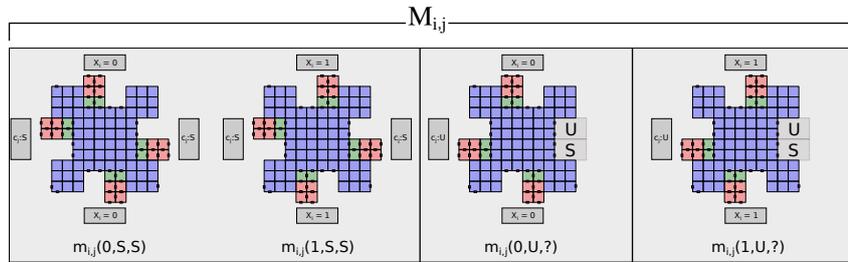
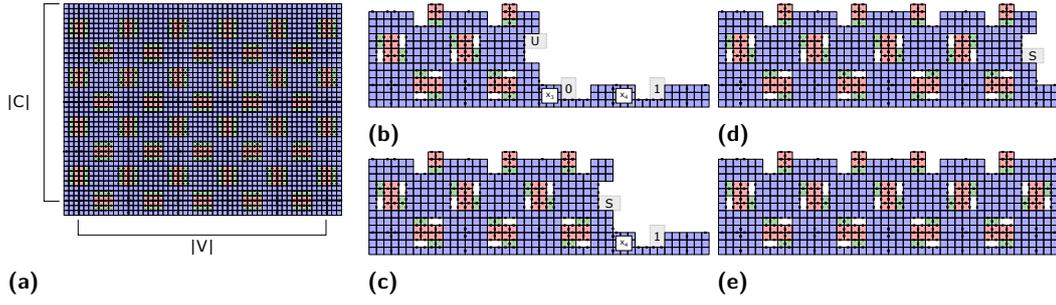


Figure 3 Possible Macroblocks that make up $\mathcal{M}_{i,j}$. Arm positions represent the value assigned to x_i and whether or not c_j has been satisfied. There will always be 4 macroblocks in each set. The left pair of macroblocks are always included and will attach if a clause is already satisfied. The remaining macroblocks attach if the clause is not yet satisfied, and their arm positions depend on ϕ . If the positive literal x_i is in c_j , $m_{i,j}(1,U,S) \in \mathcal{M}_{i,j}$, otherwise $m_{i,j}(1,U,U) \in \mathcal{M}_{i,j}$. If the negative literal \bar{x}_i is in c_j , $m_{i,j}(0,U,S) \in \mathcal{M}_{i,j}$, otherwise $m_{i,j}(0,U,U) \in \mathcal{M}_{i,j}$.

(green and red tiles in the figures). As shown in Figure 1c, each arm position represents a certain value. For vertical arms, the position represents either 0 or 1. For horizontal arms, their position is either U or S , meaning unsatisfied or satisfied.

These assemblies will combine to form an assembly for each possible assignment to ϕ (Figure 2a). We include a unique left edge assembly for each clause whose arm is always in the top position representing that the clause is currently not yet satisfied. A left edge assembly can attach to an assembly that encodes an assignment to ϕ . This starts to form an L shaped frame assembly as shown in Figure 2b. Macroblocks may attach to this frame if it has complementary arms to the currently growing frame assembly. As shown in Figure 2c, a macroblock can attach using two strength-1 glues on its east and south sides. If the arms have complimentary positions the attachment will be able to take place. However, if the arms overlaps, the macroblock is geometrically blocked from attaching, and thus not allowed.

The final challenge for designing this reduction is there must exist a single assembly that is produced regardless of the satisfying assignment. This means we must hide the values passed between macroblocks. We do this by including a certain subset of the tiles which will fill any spaces left between macroblocks.



■ **Figure 4** (a) Target assembly for producibility in the 2HAM with prebuilt assemblies. Target assembly is a $10|C|+3$ by $10|V|+6$ rectangle. Smaller rectangles between tiles denote strength-1 glues. Glues between blue tiles are not shown. Each blue tile shares a strength-2 glue with neighboring blue tiles. The exceptions are tiles separated by the thicker borders that do not share a glue unless shown. (b) A frame assembly with macroblocks attached. Here, $c_1 = \bar{x}_1 \vee x_3 \vee x_4$. (c) Here $x_3 = 0$ satisfies the clause so this macroblock that attaches has its arm in the S position. (d) The macroblocks that attach after always have their arms in the S position. (e) The right edge assembly can attach to the last macroblock since its arm position is in on the S position completing the row.

3.2 Macroblocks

A single macroblock can be seen in Figure 1b and has two parts: the body that contains glues to allow attachment (blue), and four arms which encode ϕ (green/red). Each arm on the macroblock encodes a single bit of information by being in one of two positions. We call these positions “0” and “1” for the north/south arms and “U” and “S” (unsatisfied/satisfied) for the east/west arms (Figure 1c).

We denote a macroblock representing a variable/clause pair (v_i, c_j) by its glues and arm positions as $m_{i,j}(b, w, e)$ where $b \in \{0, 1\}$ is the position of the north/south arm, $w \in \{U, S\}$ is the position of the west arm, and $e \in \{U, S\}$ is the position of the east arm. Each macroblock has a single strength-1 glue on each side (macroblocks representing the last clause do not contain glues or arms on their north side). The glues indicate which variable and clause pair this macroblock represents. The north and south glues relate to the variable, and the east and west glues relate to the clause. The south and west glues allow for cooperative attachment to an assembly that already contains macroblocks $m_{i-1,j}$ and $m_{i,j-1}$. The north and east glues allow for attachment of the next macroblocks.

Each variable/clause pair (v_i, c_j) has a set $\mathcal{M}_{i,j}$ of four macroblocks associated with it (shown in Figure 3). The exact macroblocks that are included depends on whether x_i or \bar{x}_i is present in the j^{th} clause. The macroblocks $m_{i,j}(0, S, S)$ and $m_{i,j}(1, S, S)$ are always included since the assignment of a variable can not change a clause from being satisfied to unsatisfied. If the the positive literal v_i appears in c_j , then we include the macroblocks $m_{i,j}(1, U, S)$, or $m_{i,j}(1, U, U)$ if it does not. If the negative literal \bar{v}_i appears is in c_j we include the macroblock $m_{i,j}(0, U, S)$, or $m_{i,j}(0, U, U)$ if it does not.

3.3 Computing Clauses

The left edge assembly starts with an arm in the U position. Macroblocks maintain this position (Figure 4b) until a macroblock attaches that satisfies the clause, and changes the arm to an S position (Figure 4c). Once a row of the assembly is complete (Figure 4d), if the horizontal arm is in the satisfied position, the right edge assembly can attach cooperatively and complete the row (Figure 4e). For a right edge assembly to attach, the clause must be

satisfied and the assembly below it (either another right edge assembly or the right corner assembly) must attach as well. Thus, the right edge assembly only attaches if the clause it represents is satisfied.

Each row has multiple size-2 holes between macroblocks. Filler tiles cannot attach to macroblocks on their own due to the temperature of the system. However, once two macroblocks are attached, the tiles can cooperatively bind across the hole with strength-1 glues from each side (Figure 5b). As shown in Figure 5c, this hides the previously used arm positions. The exposed northern arms also continue to encode the input string so the next clause can be computed after the attachment of the next left edge assembly.

► **Theorem 3.** *The producibility problem in the 2HAM with prebuilt assemblies of size $\mathcal{O}(1)$ is NP-complete with $\tau = 2$ and an initial assembly set containing $\mathcal{O}(|V||C|)$ distinct assemblies.*

Proof. For membership in the class NP consider an assembly tree v for a producible assembly A . To verify A is producible we may use v as a certificate. If v is a valid tree (each combination is legal) and each leaf is in the input set I , then A is producible.

To show NP-hardness we reduce from 3SAT. Given a formula ϕ in 3CNF form with $|V|$ variables and $|C|$ clauses. Our target assembly A is the rectangle described above made from macroblocks and edge assemblies with each of the spaces between arms completely filled with tiles. The input assembly set includes I , $|C|$ left edge assemblies with their arm in the unsatisfied position, and $|C|$ right edge assemblies in their satisfied position. For each variable, two input assemblies representing 0 and 1 assignments are included. The clauses are encoded by the selection of macroblock arm combinations. A set of 4 macroblocks are included in I for each variable and clause combination, so there are a total of $\mathcal{O}(|V||C|)$ macroblocks. This results in a total initial assembly set size of $\mathcal{O}(|V||C|)$, and each assembly is constant sized.

The starting assemblies, I , will grow into A if and only if ϕ is satisfiable. If ϕ is satisfiable by some assignment x , then the ‘L’ shaped frame assembly representing x will grow by attaching macroblocks. Since x satisfies ϕ each clause will eventually have its arm position changed to satisfied allowing for all the right edge assemblies to attach. The single tiles will then fill in the spaces to complete A . If A is producible then there exists some ‘L’ shaped frame assembled that grew into A . The only way this frame could have grown into A is if the position of the arms on the input assemblies represented a string x that satisfied each clause meaning ϕ is satisfiable. ◀

4 Unique Assembly Verification is coNP^{NP} -complete

In this section we show coNP^{NP} -completeness of the Unique Assembly verification problem in the 2HAM with constant sized prebuilt assemblies. We start by proving UAV is in the class of problems solvable by a nondeterministic algorithm with access to an oracle for a problem in the class NP. We then show hardness by reducing from $\forall\exists\text{SAT}$. This is an extension of the reduction shown in the previous section, and further, we utilize similar techniques used in previous reductions in the 2HAM and Tile Automata [2, 16].

► **Lemma 4.** *The Unique Assembly Verification problem in the 2HAM with prebuilt assemblies is in coNP^{NP}*

Proof. Given an instance (Γ, A) , refer to a “rogue assembly” R as a producible assembly in the system Γ that is either (1) not a subassembly of the target assembly A , $R \not\sqsubseteq A$, or (2) a strict subassembly of A and terminal, i.e., $R \sqsubset A$ and $R \in P_{(S,\tau)}$. The following nondeterministic algorithm solves UAV.

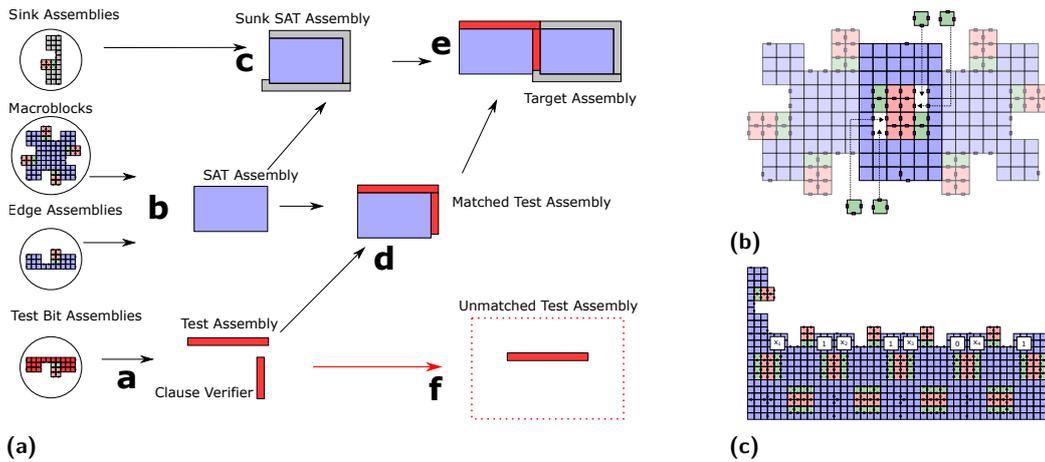


Figure 5 (a.a) Test bit assemblies come together to build a test assembly for all possible assignments of the variables in X . Clause Verifier assemblies may attach to SAT assemblies that satisfied all clauses. (a.b) Macroblocks and edge assemblies from the previous reduction are used to create SAT assemblies that evaluate the formula for every assignment of all the variables. (a.c) The sink assemblies begin attaching to SAT assemblies, ensuring they all grow into the target assembly. (a.d) A test assembly will attach to a SAT assembly that satisfies the formula and has matching assignments to the variables in X . (a.e) Matched test assemblies and sunk SAT assemblies attach to each other forming the target assembly. (a.f) Any test assembly that does not find a SAT assembly to attach to is unmatched and terminal. If any unmatched test assemblies exist, the instance of UAV is false. (b) Once two macroblocks attach, the green filler tiles are able to cooperatively attach using one glue on the macroblock, and the other glue from the red tiles of the arms from the other macroblock. The filling process hides the information that was passed. (c) Another left edge assembly may attach above the first. Since the north arms of macroblocks encode the variable assignment, the second clause may be computed in the same way as the first.

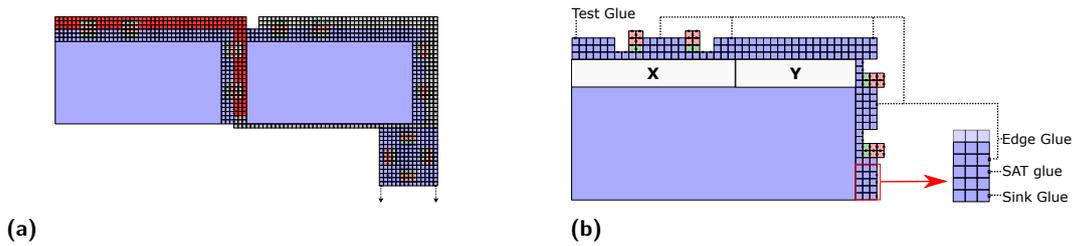
1. Nondeterministically build an assembly B of size $|B| \leq 2|A|$.
2. If B is a rogue assembly, reject.

It suffices to check all assemblies B up to size $2|A|$ since any assembly of size $> 2|A|$ must have been built from at least one other assembly B' , s.t. $|A| < |B'| \leq 2|A|$. B' is a rogue assembly itself and will be accounted for in a different branch of the computation. It remains to be checked whether B is a rogue assembly. The first condition can be verified in polynomial time by checking if B is a subassembly of A . The second condition can be checked using an NP oracle that answers the following: “Does there exist an assembly C , $|C| \leq |A|$, such that C can attach to B ?”. This problem can be solved by an NP machine that nondeterministically builds an assembly C up to size A and attempts to attach it to B . If any C can attach to B , B is not terminal. If any branch finds a rogue assembly, the co-nondeterministic machine will reject. ◀

4.1 Reduction Overview

► **Definition 5** ($\forall\text{ESAT}$). *Given an n -bit Boolean formula $\phi(x_1, x_2, \dots, x_n)$ with the inputs divided into two sets X and Y , for every assignment to X , does there exist an assignment to Y such that $\phi(X, Y) = 1$?*

We show this problem is coNP^{NP} -hard by reducing from $\forall\text{ESAT}$. An overview of the important assemblies and processes are shown in Figure 5a. The same construction used in the previous reduction is used to create exponentially many “SAT assemblies”, each of



■ **Figure 6** (a) Target Assembly for UAV. Assembly can be divided into three parts, Test assembly with satisfied SAT assembly, Sunk SAT assembly, and a column of clean up frames (b) SAT assembly. Built using the previous reduction with additional northern arms for the variables in X . Also we do not add right edge assemblies an arm is exposed which shows whether a clause has been satisfied.

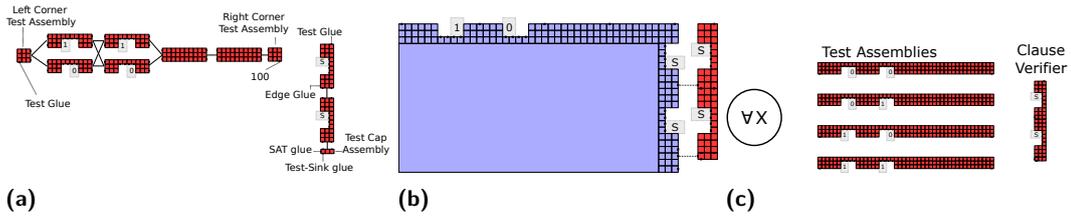
which evaluates the Boolean formula on one of its input assignments. A SAT assembly is shown in Figure 6b. We do not include right edge assemblies so SAT assemblies that have finished computing will have exposed arms on their right side denoting whether or not each clause was satisfied. The assembly has exposed arms to the north above variables in X that represent their assigned values. Variables in Y will still have no arms on their north side. We construct a test assembly (Figure 7a) for each possible assignment to the variables in X (Figure 8a) along with a *clause verifier* assembly. The clause verifier assemblies are made of right edge assemblies with their arms in the S position. Each test assembly can attach to any SAT assembly with matching assignments to X (complimentary arm positions) if the clause verifier has already attached. Other assemblies are included that “sink” all assemblies to the target except for test assemblies that did not attach to a SAT assembly. Test assemblies may build into the target assembly if and only if they find a matching SAT assembly. The key question about the system is “For all test assemblies, does there exist a compatible SAT assembly where all clauses evaluated to true?”

This system uniquely constructs the target assembly if and only if the instance of $\forall \exists \text{SAT}$ is true. If the instance is false, there exists an assignment to the variables in X where no satisfying assignment of Y exists. In this case, the test assembly representing that assignment will be terminal, which means the system does not uniquely produce the target assembly.

4.2 SAT Assembly

We use the assemblies from the previous reduction to compute all satisfying assignments to ϕ . We use the same macroblocks, input assemblies, and left edge assemblies, however, we do not include any right edge assemblies or the right corner assembly. A frame assembly is constructed for each assignment to ϕ . The assembly process then computes whether or not the assignment satisfies the clauses in the same way by attaching macroblocks with matching geometry. In this construction we mark the assignment to variables in X by including northern arms to the top most macroblocks for those variables instead of omitting them as in the previous construction. This results in a final assembly that has its assignment to X and the computed value of each clause being encoded in the exposed arm positions.

The assembly can be seen in Figure 6b with glues labeled. The exposed glues all have strength-2. In the bottom right corner of the assembly, the last variable assembly has two glues. The bottom glue is called the sink glue and this is one of the glues the sink assembly uses to attach to a SAT assembly. The glue above it, called the SAT glue, is used by both the left sink base assembly and the test assembly. The next glue appears on each macroblock with an exposed arm, and the northern side of the rightmost macroblock. This glue allows



■ **Figure 7** (a) Test bit assemblies nondeterministically construct a test assembly for each assignment to the variables in X . Right edge assemblies with arms in the satisfied positions are used to build the clause verifier. Only the north most edge assembly has an edge glue to allow a clause verifier to attach to a SAT assembly. A test assembly may attach to a SAT assembly with a clause checker attached using the test glues on their south side. The Test-Sink glue is used to cooperatively attach to a sink assembly once the test assembly is matched with a SAT assembly. (b) A clause verifier may attach to SAT assemblies that have their arms in the satisfied position. (c) Test Assemblies that are created for a X that contains 2 variables along with the single clause verifier.

for sink assemblies to attach and cover exposed arms to reach the target assembly. The test glue is the last glue on this assembly and appears in the top right corner. The test assembly uses this glue with the SAT glue to attach to a SAT assembly with matching geometry.

4.3 Test Assembly

A test assembly is constructed for each possible assignment to X starting from constant sized test bit assemblies shown in Figure 7a. For each variable in X , we include two test bit assemblies. For each each variable in Y , we include a blank test bit assembly, which is a 3×10 rectangle. This row is capped by left and right corner test assemblies. The left and right corner assemblies have a strength-1 glue. The clause verifier assembly is built using right edge assemblies. These assemblies all have their arms in the satisfied position. The north most assembly has a strength-1 glue on its left side to attach to SAT assemblies and another on its north side to allow test assemblies to attach. They connect downward to a 1×3 test cap assembly. The test cap has the strength-1 SAT glue on its left side to allow a test assembly to cooperatively bind to a SAT assembly with all arms in the satisfied position (Figure 7b). On its south side it has the test-sink glue which will be used to attach to a sink assembly cooperatively once a SAT assembly is found. This assembly process creates a test assembly for each possible assignment to the variables in X . The example of test assemblies for an instance where $|X| = 2$ and $|Y| = 2$ can be seen in Figure 7c.

The test assembly has two exposed glues on opposite ends of the assembly. Thus, the assembly cannot attach to a SAT assembly until it is completely constructed and a clause verifier assembly has attached. A clause verifier assembly may only attach to SAT assembly with matching arm positions, i.e., SAT assemblies that satisfied all clauses. A test assembly may only attach to SAT assemblies with a clause verifier and that have the same assignment to X . A test assembly along with the four possible SAT assemblies it could attach to is shown in Figure 8a. A test assembly is terminal if there does not exist a SAT assembly with the same assignment to X that satisfies all clauses. A terminal test assembly can be seen in 8b. We call these terminal test assemblies *unmatched* test assemblies. Since we construct a SAT assembly for each possible assignment to the formula ϕ , if the test assembly representing a partial assignment x is terminal that means there does not exist a remaining assignment to the variables in Y that satisfies the formula, and causes the instance of UAV to be false.

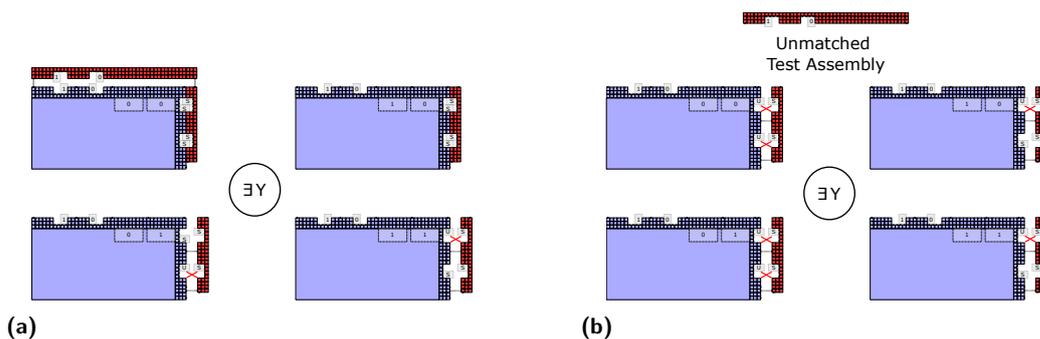


Figure 8 (a) If there exists a satisfying assignment to Y when $X = 10$, the test assembly with those arms can attach to a SAT assembly. (b) If there does not exist a satisfying assignment, all the SAT assemblies will have at least one arm not in the S position that will geometrically block the test assembly’s arm. This test assembly will be terminal so the answer to UAV will be no.

4.4 Sink

Since our goal is to design a system that uniquely constructs an assembly when the instance of $\forall\text{SAT}$ is true, we will *sink* assemblies representing a non-satisfying assignment to the target assembly. This ensures that each assembly (besides unmatched test assemblies) must eventually grow into the target assembly thus sinking all other producible assemblies to our target. We do so via sink assemblies and macroblock frames. The sink assemblies are shown in Figure 9a. The first sink assembly, the right sink base assembly, is similar to the right corner assembly of the previous section but is of height 4. Sink base tiles are single tiles that may attach to the right base assembly on its left side bottom row, which eventually connects to the left base assembly.

The right sink base assembly attaches to any SAT assembly that has not attached to a test assembly using the sink and test-sink glues. We call the attachments that occur after this the *Sink Process*. During the Sink Process, sink edge assemblies attach cooperatively with the right sink base assembly and with the SAT assembly. This allows for the sink edge assemblies to attach one-by-one and “cover” each exposed arm on the SAT assembly. The last sink edge (northern most) is slightly longer to allow for the horizontal sink edges to attach across the top of the assembly. The left sink base assembly cooperatively attaches to test assemblies that have already attached to SAT assemblies using the SAT glue on its right side and the test-sink glue on its north side.

The last assembly type that has not been accounted for in the final assembly are any unused macroblocks. In the previous reduction, any unused macroblocks are terminal since they are never used in the computation. In this reduction, we cannot have any other terminal assembly, so these must be included in our target assembly. We do this by adding frames to store the macroblocks. For each macroblock we include in our input set, we also include a clean up frame (Figure 9b). Any macroblock is now not terminal as it can attach to the clean up frame. These frames are enumerated and attach in order to the south side of the sink assembly in a single column (Figure 9c).

► **Theorem 6.** *The Unique Assembly Verification problem in the 2HAM with prebuilt assemblies of size $\mathcal{O}(1)$ is coNP^{NP} -complete with an initial assembly set size of $\mathcal{O}(|V||C|)$ and $\tau = 2$.*

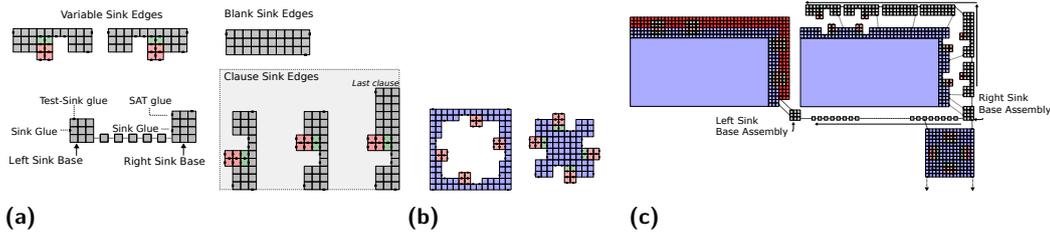


Figure 9 (a) The set of sink assemblies. The sink glue and test-sink glue are utilized for cooperative attachment. (b) For each macroblock we also include a frame so none of the macroblocks are terminal. These frames attach to each other in order at the bottom of the sink assembly. (c) The process of sinking all assemblies towards the target assembly.

Proof. We show membership in Lemma 4. Given an instance of $\forall\exists\text{SAT}$ with a formula $\phi(x_1, x_2, \dots, x_n)$ we create a 2HAM system S that uniquely assembles a target assembly S if and only if the instance of $\forall\exists\text{SAT}$ is true. If the instance of $\forall\exists\text{SAT}$ is false then S will produce a test assembly that is terminal.

The construction of SAT assemblies begins with combinations of variable and edge assemblies to produce an L shaped frame assembly for each possible assignment to ϕ . The output of ϕ on each assignment is then computed by the attachment of macroblocks that encode the clauses of ϕ , and producing a SAT assembly. The SAT assemblies expose arms representing the assignment of the variables in X , as well as arms that represent whether each clause has been satisfied. From the included prebuilt assemblies, a test assembly is produced for each possible assignment to X . Due to their arm positions, they may only attach to SAT assemblies that have a matching assignment to X and that represent an assignment that satisfies every clause.

Each assembly besides unmatched test assemblies will sink to the target assembly. Every prebuilt assembly that was designed for building a SAT assembly will be used in the construction of at least one SAT assembly, besides the macroblocks. In some cases it is possible for a macroblock to not be able to attach to any SAT assembly. To account for this this we include a macroblock frame for each macroblock to attach to, ensuring that no macroblock is terminal. The right sink base assembly may attach to any SAT assembly regardless of arm position so none of the SAT assemblies are terminal. Each sink assembly is used in the process of reaching the target assembly so none are terminal.

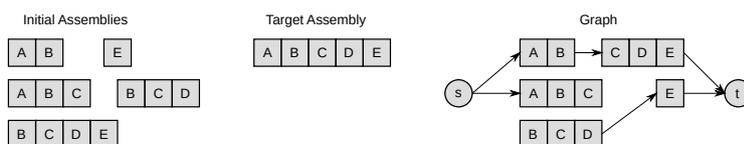
A test assembly is only terminal if there does not exist a SAT assembly with matching X arm positions that has each clause satisfied. This means for that assignment to X there does not exist an assignment to Y that satisfied ϕ , and the instance of $\forall\exists\text{SAT}$ is false.

The new assemblies in this reduction only increase the number of assemblies by a constant factor. The test and sink assemblies only add $\mathcal{O}(|V| + |C|)$ to the input assembly size, and the added macroblock frames are equal to the number of macroblocks, which is constant. \blacktriangleleft

5 1D Verification

Here, we show that the producibility and the UAV problems in one-dimensional 2HAM (all assemblies are of height-1, and tiles only have glues on their left and right sides) with prebuilt assemblies is solvable in polynomial time. Proofs omitted due to space.

Producibility Verification. At a high level, the algorithm constructs a graph of initial assemblies that may be combined to form the target assembly A . An example graph is shown in Figure 10. We add a starting node s that connects to possible leftmost assemblies and a



■ **Figure 10** Example instance of Producibility with the graph G created. The path between s and t implies a build sequence to produce the target assembly.

target node t that connects to possible rightmost assemblies. There exists an edge between two assemblies if they may attach as part of an assembly sequence building A . Thus, a path from s to t implies an assembly sequence using those nodes to build the target assembly.

► **Theorem 7.** *The producibility problem in the one-dimensional 2HAM with prebuilt assemblies is solvable in polynomial time.*

Unique Assembly Verification. Given an instance of UAV (Γ, A) , we first present three conditions that, if checked, provide the answer. If and only if all these conditions are true, the answer to UAV is “yes”: 1) A is producible. 2) There does not exist a subassembly $B \subseteq A$ that is terminal. 3) There does not exist a producible assembly $R \not\subseteq A$. Here, R is a rogue assembly since it will never grow into A . As a given system can have an exponential number of assemblies of size $\leq |A|$, we present a lemma showing a limit on the search for a witness that condition 3 is false.

► **Lemma 8.** *Let (Γ, A) be an instance of UAV in the one-dimensional 2HAM with prebuilt assemblies such that every initial assembly is a subassembly of A , and A is producible. If there exists a rogue assembly $R \not\subseteq A$, then there exists a rogue assembly R' that can be produced by combining two assemblies R'_1, R'_2 that are subassemblies of A .*

Utilizing this lemma, we search only the subassemblies of our target assembly A to verify the third condition. 1D assemblies only have a polynomial number of subassemblies, so this allows us to check both the second and third condition. Combining these two steps with the polynomial time producibility algorithm from above, we solve UAV in polynomial time.

► **Theorem 9.** *The Unique Assembly Verification problem in the one-dimensional 2HAM with prebuilt assemblies is solvable in polynomial time.*

6 Future Work

Here we showed NP-completeness of the producibility problem and coNP^{NP} -completeness of UAV in the 2HAM with constant-sized prebuilt assemblies with $\tau = 2$. The non-cooperative versions ($\tau = 1$) of many models see a drop in complexity for these problems and have been proven to be incapable of universal computation. However, the authors of [11] show that even without cooperative binding the, Polyomino Tile Assembly Model (tiles may be larger polyominoes instead of only unit squares) is capable of universal computation. Since the polyominoes in this model are similar to prebuilt macroblocks, perhaps the same results may be proven in the 2HAM with prebuilt assemblies and $\tau = 1$.

References

- 1 Leonard M. Adleman, Qi Cheng, Ashish Goel, Ming-Deh A. Huang, David Kempe, Pablo Moisset de Espanés, and Paul W. K. Rothmund. Combinatorial optimization problems in self-assembly. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pages 23–32, 2002.

- 2 David Caballero, Timothy Gomez, Robert Schweller, and Tim Wylie. Verification and Computation in Restricted Tile Automata. In Cody Geary and Matthew J. Patitz, editors, *26th International Conference on DNA Computing and Molecular Programming (DNA 26)*, volume 174 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:18, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DNA.2020.10.
- 3 Sarah Cannon, Erik D. Demaine, Martin L. Demaine, Sarah Eisenstat, Matthew J. Patitz, Robert T. Schweller, Scott M Summers, and Andrew Winslow. Two Hands Are Better Than One (up to constant factors): Self-Assembly In The 2HAM vs. aTAM. In *30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013)*, volume 20 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 172–184. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013.
- 4 Angel A Cantu, Austin Luchsinger, Robert Schweller, and Tim Wylie. Covert computation in self-assembled circuits. *Algorithmica*, 83(2):531–552, 2021.
- 5 Erik D Demaine, Martin L Demaine, Sándor P Fekete, Mashhood Ishaque, Eynat Rafalin, Robert T Schweller, and Diane L Souvaine. Staged self-assembly: nanomanufacture of arbitrary shapes with $o(1)$ glues. *Natural Computing*, 7(3):347–370, 2008.
- 6 David Doty. Theory of algorithmic self-assembly. *Communications of the ACM*, 55(12):78–88, 2012.
- 7 David Doty. Producibility in hierarchical self-assembly. In Oscar H. Ibarra, Lila Kari, and Steffen Kopecki, editors, *Unconventional Computation and Natural Computation*, pages 142–154, Cham, 2014. Springer International Publishing.
- 8 David Doty, Lila Kari, and Benoît Masson. Negative interactions in irreversible self-assembly. *Algorithmica*, 66(1):153–172, 2013.
- 9 Masayuki Endo, Tsutomu Sugita, Yousuke Katsuda, Kumi Hidaka, and Hiroshi Sugiyama. Programmed-assembly system using DNA jigsaw pieces. *Chemistry: A European Journal*, pages 5362–5368, 2010.
- 10 Constantine Evans. *Crystals that Count! Physical Principles and Experimental Investigations of DNA Tile Self-Assembly*. PhD thesis, California Inst. of Tech., 2014.
- 11 Sándor P Fekete, Jacob Hendricks, Matthew J Patitz, Trent A Rogers, and Robert T Schweller. Universal computation with arbitrary polyomino tiles in non-cooperative self-assembly. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 148–167. SIAM, 2014.
- 12 Bin Fu, Matthew J Patitz, Robert T Schweller, and Robert Sheline. Self-assembly with geometric tiles. In *International Colloquium on Automata, Languages, and Programming*, pages 714–725. Springer, 2012.
- 13 Pierre-Étienne Meunier, Damien Regnault, and Damien Woods. The program-size complexity of self-assembled paths. In *STOC: Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2020, pages 727–737. Association for Computing Machinery, 2020. doi:10.1145/3357713.3384263.
- 14 Matthew J. Patitz. An introduction to tile-based self-assembly and a survey of recent results. *Natural Computing*, 13(2):195–224, June 2014.
- 15 Robert Schweller, Andrew Winslow, and Tim Wylie. Complexities for high-temperature two-handed tile self-assembly. In Robert Brijder and Lulu Qian, editors, *DNA Computing and Molecular Programming*, pages 98–109, Cham, 2017. Springer International Publishing.
- 16 Robert Schweller, Andrew Winslow, and Tim Wylie. Verification in staged tile self-assembly. *Natural Computing*, 18(1):107–117, 2019.
- 17 Erik Winfree. *Algorithmic Self-Assembly of DNA*. PhD thesis, California Institute of Technology, June 1998.
- 18 Sungwook Woo and Paul W.K. Rothemund. Stacking bonds: Programming molecular recognition based on the geometry of DNA nanostructures. *Nature Chemistry*, 3:620–627, 2011.

- 19 Damien Woods. Intrinsic universality and the computational power of self-assembly. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 373(2046):16–22, 2013.
- 20 Damien Woods, David Doty, Cameron Myhrvold, Joy Hui, Felix Zhou, Peng Yin, and Erik Winfree. Diverse and robust molecular algorithms using reprogrammable DNA self-assembly. *Nature*, 567:366–372, March 2019. doi:10.1038/s41586-019-1014-9.

Robustness of Distances and Diameter in a Fragile Network

Arnaud Casteigts   

LaBRI, CNRS, Université de Bordeaux, Bordeaux INP, France

Timothée Corsini   

LaBRI, CNRS, Université de Bordeaux, Bordeaux INP, France

Hervé Hocquard   

LaBRI, CNRS, Université de Bordeaux, Bordeaux INP, France

Arnaud Labourel   

Aix Marseille Univ, CNRS, LIS, Marseille, France

Abstract

A property of a graph G is *robust* if it remains unchanged in all connected spanning subgraphs of G . This form of robustness is motivated by networking contexts where some links eventually fail permanently, and the network keeps being used so long as it is connected. It is then natural to ask how certain properties of the network may be impacted as the network deteriorates. In this paper, we focus on two particular properties, which are the diameter, and pairwise distances among nodes. Surprisingly, the complexities of deciding whether these properties are robust are quite different: deciding the robustness of the diameter is **coNP-complete**, whereas deciding the robustness of the distance between two given nodes has a linear time complexity. This is counterintuitive, because the diameter consists of the maximum distance over all pairs of nodes, thus one may expect that the robustness of the diameter reduces to testing the robustness of pairwise distances. On the technical side, the difficulty of the diameter is established through a reduction from hamiltonian paths. The linear time algorithm for deciding robustness of the distance relies on a new characterization of two-terminal series-parallel graphs (TTSPs) in terms of excluded rooted minor, which may be of independent interest.

2012 ACM Subject Classification Mathematics of computing → Paths and connectivity problems; Networks → Network dynamics; Theory of computation → Complexity classes

Keywords and phrases Dynamic networks, Longest path, Series-parallel graphs, Rooted minors

Digital Object Identifier 10.4230/LIPIcs.SAND.2022.9

Funding ANR project ESTATE (ANR-16-CE25-0009-03) and DREAMY (ANR-21-CE48-0003)

1 Introduction

The diameter of a network is the maximum distance between any pair of nodes. This concept plays an important role in various fields of network science. For example, in communication networks and distributed algorithms, the diameter is a key parameter involved in the complexity of basic tasks such as leader election, spanning tree construction, and broadcast. Indeed, both the execution time and number of messages may depend on this parameter. Similarly, distances between nodes play a role in nearly all networking phenomena.

In a physical network, the links may deteriorate and eventually become subject to permanent failure. In this case, either the network is maintained (repaired), or it is used despite the failures so long as communication remains possible, i.e., so long as it remains *connected*. A natural question, is then to what extent the properties of the network could change as the network deteriorates. In graph theoretical terms, the connectivity assumption imposes that the communication graph always remains a *connected spanning subgraph* of the original graph, although one does not know in advance which such subgraph will occur. A notion of robustness accounting for the preservation of a property in *all* these subgraphs



© Arnaud Casteigts, Timothée Corsini, Hervé Hocquard, and Arnaud Labourel; licensed under Creative Commons License CC-BY 4.0

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).

Editors: James Aspnes and Othon Michail; Article No. 9; pp. 9:1–9:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

was investigated in [3] in the context of covering problems. Although it can be formulated in classical graph theoretical terms, the notion of robustness was initially motivated by a temporal context. Namely, if the lifetime is infinite, then some edges may be recurrent (i.e., always reappear), and some may eventually disappear forever. If the network is guaranteed to have a recurrent temporal connectivity (class $\mathcal{TC}^{\mathcal{R}}$), then a certain connected spanning subset of the edges *must* be recurrent, although one does not know in advance which subset. (In technical terms, the *eventual footprint* is any connected spanning subgraph of the *footprint*.)

In this paper, we investigate the robustness of distances and diameter in the classical (i.e. non-temporal) setting, where deletions are definitive. Relations between edge deletions and distances in a graph have been studied over decades in some fields, such as that of graph spanners. A spanner of a graph is a subgraph that preserves, to some extent, the properties of the input graph – typically, the distances – while retaining as few edges as possible. For example, we know since [1] that a tradeoff exists between the size of the spanner and the deterioration of distances, in proportional terms (called *stretch factor*). Namely, there always exists a spanner of size $O(n^{1+1/k})$ whose stretch factor is at most $2k - 1$, where n is the number of nodes in the graph. The reader is referred to [6] for background in graph spanners. A significant difference between this topic and questions about robustness is that, in the case of spanners, the deletions of edges are *chosen* by the algorithm, whereas in the case of robustness, they are imposed by the environment. Thus, it makes sense to think of robustness in terms of adversarial edge deletions. The question is then whether and how the distances are preserved under all possible choices of deletions by an adversary, up to preserving connectivity.

As a warm-up, observe that, for any path P connecting two nodes u and v in a graph G , the adversary can always delete enough edges that this path becomes the only path between u and v (for example, by choosing a spanning tree $G' \subseteq G$ that contains this path). Thus, deciding whether the distance between u and v is robust comes to decide if there exists a path between u and v that is longer than their original distance $d(u, v)$ in G . A similar question was considered recently in the case of *induced* paths [2] and was shown to be solvable in polynomial time, albeit with a time complexity of $O(|G|^{18})$ (as of the current analysis, which the authors of [2] do not consider tight). The non-induced case is arguably simpler. In fact, the question for non-induced paths reduces (without being equivalent) to a question known as the *next-to-shortest* path, which consists of finding a path between u and v that is the shortest among all paths of cost strictly greater than $d(u, v)$. Clearly, $d(u, v)$ is robust if and only if no such path exist. A number of algorithms were introduced for this problem, both in directed [11] and undirected [10, 12, 9] graphs. The best known algorithm, in the undirected case, is that of [9], with a time complexity of $O(n^2)$. It is not known whether this algorithm is optimal for general graphs (in particular, sparse graphs), the quadratic term being independent from the number of edges.

1.1 Contributions

The first set of contributions of this paper is a structural investigation of robustness, which results in a linear time algorithm for deciding whether the distance between two vertices u and v is robust. To do so, we identify and exploit a connection between robust distances and *two-terminal series-parallel* graphs (TTSPs), whose recognition is known to be solvable in linear time [16]. Precisely, we introduce a new class of TTSP graphs, referred to as *TTSPs of fixed length*. Then, we show that the distance between u and v is robust in a graph G if and only if the subgraph of G induced by the union of all paths from u and v is a TTSP of fixed length. The main contribution is the characterization itself. First, we show that general

TTSPs correspond to the graphs that exclude a certain *rooted minor*, namely, a diamond rooted at u and v . This point of view clarifies earlier characterizations of TTSPs that essentially arrived at the same conclusion without relying explicitly on a forbidden pattern. (In the case of series-parallel graphs which are not two-terminal, such a characterization was already known in terms of forbidden K_4 [4, 5].) Then, we establish the correspondence between these forbidden rooted diamonds and the robustness of the distance between u and v . The natural consequence of our characterization is that robustness can be tested in linear time by adapting the recognition algorithm from Valdes, Tarjan, and Lawler [16] to the special case of fixed length TTSPs, with the same running time of $O(n + m)$ operations (where m is the number of edges).

The second set of contributions is the computational complexity of deciding whether the diameter of a graph is robust. Clearly, the concept of diameter is strongly related to the one of distance. However, and quite surprisingly, computing the *robustness* of the diameter turns out to be a much different (and much more difficult) problem than computing the robustness of pairwise distances. Precisely, we show that ROBUST-DIAMETER is coNP-complete (in other words, proving that the diameter of certain networks are robust is difficult). This is done through a reduction from HAMILTONIAN-PATH, where one must decide if a path of length $n - 1$ exists in a graph.

1.2 Organization of the document

The document is organized as follows. Section 2 provides the main definitions and some basic observations. Then, we establish in Section 3 that deciding whether the diameter is robust is a difficult (coNP-complete) problem. In Section 4, we investigate several aspects of the robustness of distances, in relation to two-terminal series-parallel graphs (TTSPs). The main results of this section are a new characterization of TTSPs (Section 4.2) and a linear time decision algorithm for robustness of distances (Section 4.4). Finally, we conclude with some remarks and open questions in Section 5.

2 Definitions and preliminary observations

2.1 Basic definitions

An *undirected graph* G is a pair $(V(G), E(G))$ where $V(G)$ and $E(G)$ are two disjoint sets, the set of vertices (or nodes) and the set of edges respectively. Each edge is associated with two vertices called its *endpoints*. A *loop* is an edge whose endpoints are the same vertex. If there are several edges with the same endpoints, these edges are called a *multi-edge*. An undirected graph is *simple* if it does not have loops nor multiple edges. A graph that can have multi-edges is called a *multigraph*. A graph without loops is called *loopless*. Unless otherwise mentioned, all the graphs in this paper are simple. The *order* of a graph G is $|V(G)|$ and its *size* is $|E(G)|$. Two vertices u and v are *adjacent* if there exist an edge uv in $E(G)$. In a loopless graph, the *degree* of a vertex u in G is the number of edges incident with u . In a simple graph, this corresponds to the number of vertices with which u shares an edge, called its *neighbors*. A *complete graph* is a simple graph such that every pair of vertices are neighbors, we denote the complete graph of order n by K_n .

Let H and G be two graphs. We say that H is a *subgraph* of G if and only if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. If $V(H) = V(G)$ and $E(H) \subseteq E(G)$ then H is a *spanning subgraph* of G . Let $X \subseteq V(G)$, the *induced subgraph* $G[X]$ is the subgraph of G on vertex set X and

where, for every two vertices u and v of X , $uv \in E(G[X])$ if and only if $uv \in E(G)$. Finally, we use $G - X$ as a shorthand for $G[V - X]$ if X is a set of vertices, and for $(V, E - X)$ if X is a set of edges.

A *path* \mathcal{P} from v_0 to v_k is a sequence of edges (e_1, e_2, \dots, e_k) of length k for which there is a sequence of vertices (v_0, v_1, \dots, v_k) such that the endpoints of e_i are v_{i-1} and v_i for $i = 1, \dots, k$. All vertices in a path must be distinct, except possibly for the first and last, in which case the path is a *cycle*. In a simple graph, the sequence of vertices identifies uniquely the corresponding path. The *length* of a path is the number of edges in it. The *distance* between two vertices u and v in a graph G , denoted by $d_G(u, v)$ (or simply $d(u, v)$ when the context of G is clear), is the minimum length of a path from u to v . The *diameter* of a graph G , denoted by $\text{diam}(G)$, is the maximum distance between any pair of vertices. A graph is *connected* if for every pair of vertices u and v , there exists a path from u to v , and it is *biconnected* if for any $v \in V$, the graph $G - \{v\}$ is connected. A *tree* is a connected graph without cycle.

A *connected component* is a maximal connected subgraph. A *block* or *biconnected component* is a maximal biconnected subgraph. A *separator* of a connected graph G is a set of vertices whose removal renders G disconnected. An *articulation point* of a connected graph G is a separator of size 1 (a single vertex). The structure of blocks and separators of a connected graph can be described by a tree called the *block-cut tree* [7]. This tree has a vertex for each block and for each articulation point of the given graph. There is an edge in the block-cut tree for each block and for each articulation point that belongs to that block. For a graph G and two vertices u and v , we say that G is a *block-cut* (u, v) -*path* if the block-cut tree of G is a path from s to t , such that u (resp. v) is only contained in the block associated to s (resp. t). Observe that by definition, u and v are not articulation points.

Let us now define the notion of robustness that we consider in this work.

► **Definition 1 (Robustness).** *A property P of a connected graph G is called robust [3] if P is satisfied in every connected spanning subgraph of G (including G itself). By extension, if P denotes a quantity rather than a predicate (such as, here, a distance), then it is called robust if its value is the same in all connected spanning subgraphs of G .*

We are now ready to state the main two problems that we address in this paper, namely ROBUST-DIAMETER and ROBUST-DISTANCE.

► **Definition 2 (ROBUST-DIAMETER).**

Input: A graph G .

Output: Whether the diameter of G is robust.

► **Definition 3 (ROBUST-DISTANCE).**

Input: A graph G , two vertices u and v of G .

Output: Whether the distance between u and v is robust in G .

2.2 Preliminary observations

In this section, we establish a number of basic facts, most of which are used in the rest of the paper.

► **Lemma 4.** *Let G be a connected graph and $P \subseteq G$ a path between distinct vertices u and v , then there exist a spanning tree $S \subseteq G$ such that $P \subseteq S$.*

Proof. Observe that P is a (possibly non-spanning) tree. As long as it is not spanning, one can extend it by adding a node that is not in it, but that has a neighbor in it. Such a node always exists because G is connected. ◀

► **Lemma 5.** *Let G be a connected graph and u, v two vertices of G . The distance between u and v is robust if and only if there is no path between u and v longer than $d_G(u, v)$.*

Proof. Suppose there is a longer path in G between u and v . By Lemma 4, it is thus possible for the adversary to reduce G to a spanning tree $S \subseteq G$ such that $d_S(u, v) > d_G(u, v)$. ◀

► **Lemma 6.** *Let G be a biconnected graph and u, v and w three vertices of G . There is a path from u to w passing through v .*

Proof. Consider a graph G' obtained from G by adding an extra vertex z adjacent to both u and w . G' is also biconnected since no articulation point was added. By Menger's theorem [13], there exist two vertex-disjoint paths from z to v . Since z has only two neighbors u and w , there are two subpaths: one from u to v and another from v to w . By composing these subpaths, one can thus obtain a path from u to w passing through v . ◀

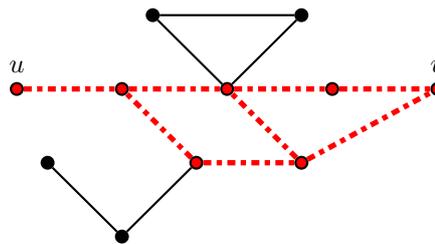
The following lemma allows us to subsequently restrict our attention to a limited part of the graph, when investigating the robustness of distance between two given vertices.

► **Lemma 7.** *Let G be a connected graph, u, v two distinct vertices of G and H the graph induced by the paths from u to v . The distance from u to v is robust in G if and only if it is robust in H .*

Proof. \implies If the distance is robust in G , since H is a subgraph of G , the distance must be robust in H .

\impliedby By definition of H , all paths from u to v in G are also in H , thus, if all those paths have length $d(u, v)$ (equivalent to the robustness of the distance), then all the paths in G from u to v have the same length and the distance is robust as well in G . ◀

Figure 1 shows a graph where the distance between u and v is not robust, this can be verified by only considering the graph induced by all paths from u to v thanks to Lemma 7. Observe that H , the graph induced by the paths from u to v , is always a block-cut (u, v) -path.



■ **Figure 1** A graph and its subgraph induced by the paths from u to v (in dashed red). This subgraph is a block-cut (u, v) -path.

► **Lemma 8.** *Let G be a graph and u, v two vertices of G . Then, the graph H induced by the paths from u to v is a block-cut (u, v) -path.*

Proof. By definition of H , all vertices in $V(H)$ are part of a path between u and v and H is connected. Let T be the block-cut tree of H , by definition, T has a vertex for each block and each articulation point of H . The leaves of T (vertices of degree 1) are blocks, the neighbors of blocks are articulation points, and the neighbors of articulation points are blocks. Suppose that H is not a block-cut (u, v) -path, it means one of the following cases:

- u or v are not part of the leaves in T . Consider a path in T (x_0, \dots, x_k) that contains both u and v (or their blocks), $u \in x_i$ and $v \in x_j$ such that $i > 0$ or $j < k$. If $i > 0$ (the same could be done with v if $j < k$), then there is a path (x_0, \dots, x_i) crossing at least one articulation point of H (v_1). It can be deduced that any vertex in v_0 cannot be part of a path between u and v which leads to a contradiction in H .
- T is a tree with at least one vertex of degree 3 or more. Since u and v must be in the leaves of T , then there is a path between u and v that crosses a vertex x of degree at least 3. Consider a neighbor y of x not part of such path. Either y is a block, and x is an articulation point and all paths from u to v in H cannot reach a vertex in y , or y is an articulation point, and there must be another block z that cannot be crossed by paths between u and v (which we assume is not the case). ◀

The following two lemmas are not used in this paper. However, they establish some connections between the robustness of distances and that of the diameter, which is of general interest in the present study.

► **Lemma 9.** *All the distances are robust in G if and only if G is a tree.*

Proof. If G is a tree, then the adversary cannot remove any edge, so all the distances are trivially robust. If it is not a tree, then at least one edge uv can be removed, and the distance between u and v thus increases from one to something strictly larger. ◀

► **Lemma 10.** *Let G be a connected graph, if $\text{diam}(G)$ is robust, then for every pair of vertices u, v in G such that $d(u, v) = \text{diam}(G)$, their distance is robust.*

Proof. By contradiction, if their distance is not robust, then there must exist a path of length greater than $d_G(u, v)$. By Lemma 4, the adversary can obtain a spanning tree containing this path, whose diameter is thus also greater than $d_G(u, v) = \text{diam}(G)$. ◀

3 Robustness of the diameter is hard

In this section, we prove that the problem of deciding whether the diameter of a graph G is robust is coNP-complete. We start with two basic facts that will be used only in this section.

► **Lemma 11.** *If H is a connected spanning subgraph of G , then $\text{diam}(H) \geq \text{diam}(G)$.*

Proof. Let H be a connected spanning subgraph of G . If $\text{diam}(H) < \text{diam}(G)$, then there must exist two vertices u, v such that $d_G(u, v) = \text{diam}(G)$ and $d_H(u, v) < \text{diam}(G)$. Let P be a path of length $d_H(u, v)$ in H . Since $H \subseteq G$, P must also exist in G , which contradicts the fact that $d_G(u, v) > d_H(u, v)$. ◀

► **Lemma 12.** *The diameter of a connected graph G is robust if and only if it is equal to the length of the longest path of G .*

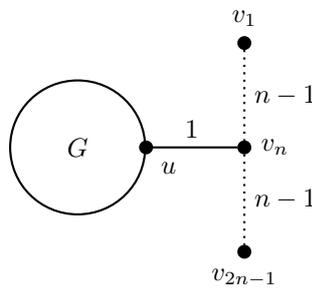
Proof. \Leftarrow Let G be a graph of diameter d that is also the longest path in G . Any connected spanning subgraph H of G has, by Lemma 11, $\text{diam}(H) \geq d$. If $\text{diam}(H) > d$, then there is a path in H (and in G) that is longer than d , which is impossible, thus $\text{diam}(H) = \text{diam}(G) = d$ for any of these graphs and the diameter of G is robust.

\Rightarrow By contradiction, let G be a graph whose diameter d is robust even though a longest path of length $l > \text{diam}(G)$ exists between some vertices u and v . By Lemma 4, the adversary can obtain a spanning tree T of G containing this path, whose diameter must be strictly larger than that of G . ◀

► **Theorem 13.** ROBUST-DIAMETER is coNP-complete.

Proof. To prove this statement, we will show that the problem is in coNP and that the HAMILTONIAN-PATH problem reduces to it in polynomial time. (HAMILTONIAN-PATH consists of deciding whether a given graph G admits a path of length $n - 1$.) The fact that ROBUST-DIAMETER is in coNP is direct, using any path of length longer than the diameter as (negative) certificate.

Now, let G be an input graph for HAMILTONIAN-PATH. Without loss of generality, we suppose that G is connected and that it is not itself a path, as otherwise the answer is trivially positive. From G , we can construct a graph H_u as follows: Let P be a graph that consists of a single path of length $2n - 2$ on vertices $\{v_i\}, i \in [1, 2n - 1]$. The graph H_u is built by picking a vertex u in $V(G)$ and adding an edge between u and v_n , the middle vertex of P_{2n-1} . See Figure 2 for an illustration.



■ **Figure 2** The graph H_u .

We will now prove that G admits a path of length $n - 1$ if and only if the diameter of H_u is not robust, for some choice of u . Since G is not itself a path, the diameter of H_u must be $2n - 2$. If the diameter of H_u is not robust for some u , then there must exist a path of length at least $2n - 1$ in some connected spanning subgraph of H_u . The only way this can happen is that $n - 1$ vertices on this path are in G , which implies that G admits a hamiltonian path (starting at u). Conversely, if G admits such a path, then there exists a choice of u such that this path will cause the diameter of H_u to be non-robust. Clearly, the above construction can be made in polynomial time, and guessing u will only contribute an additional factor of n to the complexity. ◀

4 Robustness of pairwise distances

In this section, we investigate the problem of deciding whether the distance between two vertices u and v is robust in a given graph G (ROBUST-DISTANCE problem). It turns out that the positive instances to this problem can be characterized in terms of two-terminal series-parallel graphs of a certain type. Thus, we start by defining, in Section 4.1, some basic concepts related to two-terminal series-parallel graphs (TTSPs). Our main technical contribution, described in Section 4.2, is an original characterization of TTSPs in terms of excluded rooted diamonds whose “roots” (endpoints) are u and v . This characterization may be of independent interest. In the context of ROBUST-DISTANCE, it allows us to formulate a necessary condition for the positive instances of the problem, in terms of excluding (u, v) -rooted diamonds (Section 4.3). This condition is however not sufficient, as some TTSPs with respect to u and v may admit paths of different length. We show that existing TTSP

recognition algorithms can be adapted at essentially no cost in order to test for the special case of fixed length TTSPs, which capture exactly the properties that should be tested (Section 4.4).

4.1 Two-terminal series-parallel graphs (TTSPs)

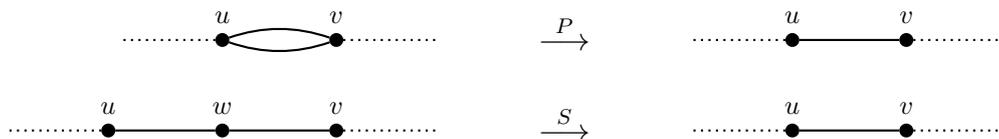
The concept of a two-terminal series-parallel graph seems to have been introduced by Riordan and Shannon in [14] (1942). It is now classically defined as follows.

► **Definition 14** (Two-terminal series-parallel graph). *Let G be a connected multigraph, s and t two distinct vertices of G called source and sink respectively. G is two-terminal series-parallel (TTSP) if it can be turned into K_2 by a sequence of the following operations:*

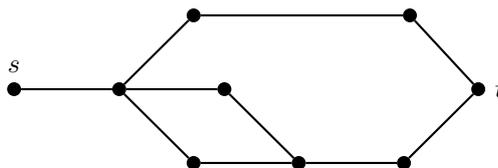
- *S : Delete a vertex of degree 2 (other than s or t) and connect its neighbors with an edge.*
- *P : Replace a pair of parallel edges with a single edge connecting the same endpoints.*

Symmetrically, TTSPs can be seen as the graphs which can be obtained from K_2 through the reverse operations of P and S .

These operations S and P are illustrated in Figure 3 and an example of TTSP graph is given in Figure 4. In this example, the distance between s and t is not robust. Note that the fact that s and t are fixed is an important aspect of TTSP graph. For example, if s and t were chosen differently in the graph of Figure 4, the graph would not be a TTSP. The class of graphs that admit a valid pair (s, t) resulting in a TTSP is called SP (for series-parallel). We do not use it in this paper.



■ **Figure 3** The operations P and S to define TTSP.



■ **Figure 4** A TTSP between s and t .

When a graph (together with a pair (s, t)) is not a TTSP, then the repeated application of rules S and P eventually fails and one is left with an irreducible graph.

► **Definition 15** (TTSP-irreducible). *Let G be a graph and u, v two vertices, then G is TTSP(u, v)-irreducible if G has at least three vertices and the operations S and P cannot be applied relative to u and v .*

The following lemma makes a connection between a TTSP(s, t) and a block-cut (s, t) -path.

► **Lemma 16** (Lemma 8 in [5]). *Let G be a TTSP graph with respect to (s, t) , then G is a block-cut (s, t) -path.*

However, the converse is not true, and some graphs that are block-cut (s, t) -paths are not a TTSP (s, t) . These graphs have special properties characterized through the following lemma (which will be used later).

► **Lemma 17.** *Let G be a multigraph that is a block-cut (u, v) path, G is TTSP (u, v) -irreducible if and only if G is simple, with at least 4 vertices, and such that for all $w \in V(G) - \{u, v\}$, $\deg(w) \geq 3$.*

Proof. Let G be such a graph.

- P cannot be applied, unless G has multiple edges, so a TTSP (u, v) -irreducible multigraph must be simple.
- S cannot be applied to G , unless there exists a vertex of degree two (other than u and v). Thus, no vertex in an irreducible graph can have degree 2, and since G is a block-cut (u, v) -path, every vertex except u and v must have degree at least 3.
- If G is TTSP (u, v) -irreducible, then it has at least 3 vertices. But since one of them has degree 3 and G is simple, then G actually has at least 4 vertices.

Conversely, if G is simple, with degree at least 3, and has at least four vertices, then (respectively), P cannot be applied, S cannot be applied, and G has at least three vertices. ◀

4.2 Characterization of TTSPs in terms of excluded rooted minor

In this section, we characterize graphs that are TTSP via an excluded rooted minor that corresponds to a complete graph of order four minus one edge called diamond. A similar characterization was mentioned in [16] in which it is stated that a directed graph is not TTSP if only if it has as a subgraph a subdivision of a directed diamond. This result was given as an easy deduction from the classical result of [4] that states that graph is not series-parallel if and only if it has as a subgraph a subdivision of K_4 . This characterization of TTSP graphs is not sufficient to directly show that the distance between the two terminal is not robust if the graph between u and v is not a TTSP for which one needs to root the terminal vertices in the minor. Moreover, the setting was different since it considers directed graph. For all these reasons, it seems worth characterizing TTSPs in terms of a clear excluded pattern, which is the purpose of this section. Let us start with basic definitions.

► **Definition 18 (Minor).** *Let G and H be two graphs, G has a minor H if there is a graph isomorphic to H from G after a succession of the following operations:*

- deleting a vertex v ;
- deleting an edge e ;
- contracting an edge xy into the vertex x : removing y and adding a new edge xz for every z such that $yz \in E(G)$.

The notion of minor is not precise enough to guarantee the non-robustness of the distance between two vertices u and v , because the position of u and v within the minor matters. Therefore, we use the finer concept of rooted minors, where some vertices can be distinguished in the minor. The difference to “normal” minors is that we want to keep a set $X \subseteq V(G)$ of root vertices alive in the minors. An X -legal minor operation is either the deletion of a vertex $y \notin X$, the deletion of any edge, or the contraction of an edge xy into x with $y \notin X$.

► **Definition 19 (Rooted minor).** *Let G and H be two graphs, $X \subseteq V(G)$ with $|X| \leq |V(H)|$, $\pi : X \rightarrow V(H)$ an injection. The pair (G, X) is said to have a π -rooted-minor if G has a minor H such that each vertex $x \in X$ corresponds to the vertex $\pi(x)$ in H obtained with X -legal minor operations.*

We are now ready to show the main technical part of this section. Observe that our definition of rooted minors differs from the definition found in [15] and [17], since all the vertices of the minor are not necessarily rooted. Let H_d be the complete graph of order four minus one edge between x and y ($K_4 \setminus \{x, y\}$). For a TTSP(u, v)-irreducible graph G , we define the bijection $\pi_d : \{u, v\} \rightarrow \{x, y\}$ by $\pi_d(u) = x$ and $\pi_d(v) = y$.

► **Lemma 20.** *If G is a TTSP(u, v)-irreducible graph and a block-cut (u, v)-path for some $u, v \in V(G)$, then G has a π_d -rooted-minor H_d .*

Proof. We will show the property by induction on the order and the size of the graph. Consider a block-cut (u, v)-path G , such graph is connected. If the order of G is less or equal to 3, then by Lemma 17, G is not a TTSP(u, v)-irreducible graph and the property is satisfied. For $n = 4$, by Lemma 17, G must have two vertices distinct from u and v with degree 3 in order to be TTSP(u, v)-irreducible, thus G has a diamond subgraph and a π_d -rooted-minor H_d . For $n > 4$, consider that G is a TTSP(u, v)-irreducible graph and a block-cut (u, v)-path of order n and size m . Assume by induction that the property is verified for all graphs of order less or equal to $n - 1$ or graphs of order n with a size less or equal to $m - 1$.

If G is not biconnected, then let c_1, c_2, \dots, c_k be the articulation points from u to v . Since G is a block-cut (u, v)-path, any path from u to v crosses these articulation points in order. Let $u = c_0$ and $v = c_{k+1}$. Observe that, for any $0 \leq i \leq k$, the block B between c_i and c_{i+1} must be a TTSP(c_i, c_{i+1})-irreducible graph since otherwise G would not be a TTSP(u, v)-irreducible graph. B is a TTSP(c_i, c_{i+1})-irreducible graph and a block-cut (c_i, c_{i+1})-path of order less than n . By induction, B has a π'_d -rooted-minor H_d with $\pi'_d : \{c_i, c_{i+1}\} \rightarrow \{x, y\}$. One can find two disjoint paths: one from u to c_i and another one from c_{i+1} to v that do not contain edges of B . It follows that G has a π_d -rooted-minor H_d . Hence, for the remainder of the proof, one can assume that G is a biconnected graph.

We now consider several cases depending on the neighborhood of u and v . Observe that the degrees of u and v must be at least two since otherwise G would not be biconnected.

■ **Case 1:** u and v are adjacent.

In this case, we consider the graph G' which is G minus the edge uv . First, we show that G' is a block-cut (u, v)-path. By Lemma 6, for any vertex $w \in V(G)$ there is a path from u to v passing through w since G is biconnected. This path also exists in G' since it does not use the edge uv in G . It follows that there is no articulation point separating w from both u and v and so G' is a block-cut (u, v)-path. Assume, by way of contradiction, that G' is a TTSP(u, v) graph. It means that there is a sequence of operations P and S such that G' can be turned into K_2 while preserving u and v . Using the same sequence of operations, G can be turned into a multigraph of two vertices u and v with two edges linking u and v . By applying an operation P , we obtain a K_2 and thus there is a contradiction with the fact that G is a TTSP(u, v)-irreducible graph. Hence, G' is a TTSP(u, v)-irreducible graph and a block-cut (u, v)-path. By induction, since G' is of order n and size $m - 1$, G' has a π_d -rooted-minor H_d and so has G .

■ **Case 2:** u is adjacent to $w \neq v$ such that w is not adjacent to other neighbors of u .

In this case, one can contract edge uw into u . Observe that if $\{u, w\}$ is a separator of G then u is the only articulation point in the connected new graph. One only keeps the block containing v to obtain the graph G' . G' is a biconnected simple graph and all of its vertices are of degree at least 3 except u and v which have degree at least 2 since G' is biconnected. By Lemma 17, G' is a TTSP(u, v)-irreducible graph. Since its order is less than n , it has a π_d -rooted-minor H_d and so has G .

- **Case 3:** u is adjacent to two vertices $w \neq v$ and $z \neq v$ that are adjacent.

We remove u from G obtaining graph G' . Since G is biconnected, G' is connected. Hence, there is a path P_{wv} from w to v in G' and a path P_{zv} from z to v . Consider a path P_{wv} without z and a path P_{zv} without w if such paths exist. If P_{wv} does not contain z and P_{zv} does not contain w then u, w, z, v define a π_d -rooted-minor H_d . If P_{wv} contains z then its subpath from z to v is a path not containing w . Since the same can be said for P_{zv} and w , it follows that either P_{wv} does not contain z or P_{zv} does not contain w . Assume, without loss of generality, that all paths between w and v contain z . It means that z is an articulation point of G' separating w and v and $\{u, z\}$ is a separator of G . Consider the subgraph G'' obtained by removing from G all vertices that are cut from v by removing $\{u, z\}$ (including w). Observe that, since G'' is biconnected, u has degree at least 2 in G'' . If z has degree 2, one contracts edge uz into u . Each other vertex of G'' has the same degree in G'' and G . It follows that all vertices of G'' have degree at least 3 except u and v that have degree at least 2. By Lemma 17, G'' is a TTSP(u, v)-irreducible graph. Since its order is less than n , it has a π_d -rooted-minor H_d and so has G . ◀

► **Proposition 21.** *TTSP(u, v) graphs correspond exactly to the block-cut (u, v)-paths which have no π_d -rooted-minor H_d .*

Proof. We show the equivalent proposition that states that G is not a TTSP(u, v) graph if and only if G is not a block-cut (u, v)-path or admits a π_d -rooted-minor H_d .

◀ By Lemma 16, if G is not a block-cut (u, v)-path then it is not a TTSP(u, v) graph. Hence, one can assume that G is a block-cut (u, v)-path and admits a π_d -rooted-minor H_d . Consider H , the graph obtained after a succession of operations S and P on G such that no more of these operations can be applied, H is either K_2 , or, by Lemma 17, a TTSP(u, v)-irreducible graph. However, S and P are $\{u, v\}$ -legal minor operation (P being an edge deletion and S being an edge contraction preserving u and v). Since H_d could not be reduced with S or P , it means that H must have a π_d -rooted-minor H_d . Therefore, H cannot be K_2 and G is not TTSP between u and v .

⇒ Suppose G is not TTSP between u and v . One can assume that G is a block-cut (u, v)-path since otherwise the property is satisfied. By Lemma 17, G can be reduced to a TTSP(u, v)-irreducible graph H . By Lemma 20, H admits a π_d -rooted-minor H_d . Since S and P are particular minor operations, G also admits a π_d -rooted-minor H_d . ◀

4.3 Robust distance in terms of rooted diamonds

With Proposition 21, we have established that any block-cut (u, v)-path that is not a TTSP must have a rooted diamond. With that characterization, it is easier to characterize the graphs in which $d(u, v)$ is not robust.

► **Lemma 22.** *Let G be a connected graph and u, v two vertices of G . If G has a π_d -rooted diamond minor in u, v , then the distance between u and v is not robust.*

Proof. Suppose G admits a π_d -rooted diamond minor in u, v , where x and y are the other two vertices (of degree 3 in the minor). It means there are four paths from u to v in G :

- c_1 that crosses x but not y ;
- c_2 that crosses y but not x ;
- c_3 that is the same path as c_1 until x , then crosses y from x before crosses the same vertices from y to v as c_2 ;
- c_4 that crosses y then x and finally v by crossing the same vertices as c_2 (until y) then c_3 (until x) then c_1 (until v).

Consider the subpaths ux , xv , xy , wy and yv crossed by the previous paths, these subpaths are all disjoint. If the distance between u and v was robust, then it would mean that $l(c_1) = l(c_2) = l(c_3) = l(c_4)$. Hence, we have $l(c_3) + l(c_4) = l(c_1) + l(c_2)$ which implies that $2l(xy) = 0$. Since x and y are distincts, a path between the two vertices must have a length of at least 1. Therefore the distance between u and v cannot be robust. ◀

► **Lemma 23.** *Let G be a block-cut (u, v) path. If G is not TTSP between u and v , then the distance between u and v is not robust.*

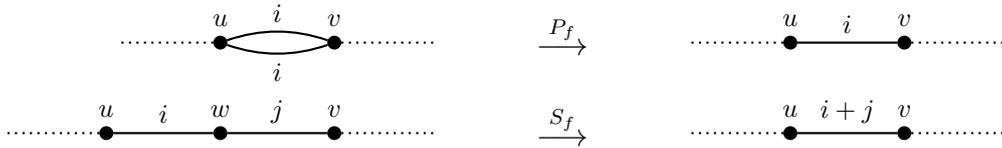
Proof. By Proposition 21, a block-cut (u, v) path which is not a TTSP (u, v) must have a rooted diamond minor in u, v . By Lemma 22, such a graph cannot have a robust distance between u and v . ◀

4.4 An efficient recognition algorithm for distance-preserving TTSPs

We are now ready to exploit the above characterizations in order to test efficiently (indeed, in linear time) whether the distance between two given vertices is robust in a given graph.

► **Definition 24.** *A graph G is a TTSP of fixed length (TTSPf) between s and t if, starting with weights of 1, it is turned into K_2 by a sequence of the following operations (see Figure 5):*

- P_f : Replace a pair of parallel edges of weight i with a single edge of weight i connecting their common endpoints.
- S_f : Replace a pair of edges of weight i and j , incident to a vertex of degree 2 other than s or t with a single edge of weight $i + j$.



■ **Figure 5** The operations P_f and S_f associated with TTSP of fixed length.

Note that a TTSP of fixed length remains *de facto* a TTSP, because the new operations are only more restricted. In the following, we say that the *length* of a weighted path corresponds to the sum of weights of its edges.

► **Lemma 25.** *Let G be a connected edge-weighted multigraph and let s and t be two distinct vertices in G . Consider the edge-weighted multigraph H that results from applying P_f and S_f exhaustively on G . For any length d , there is a path c from s to t of length d in G if and only if there is a path c' from s to t of length d in H .*

Proof. Consider two cases on H :

- If H is the result of the operation P_f on two parallel edges e_1, e_2 of weight i into an edge e' of weight i , then:
 - If c does not cross e_1 or e_2 , then c is the same in H ;
 - If e_1 or e_2 is crossed by c (but not both), then there is a path c' in H that is the same as c but crossing e' instead, c' has the same length as c . On the contrary, considering c' in H crossing e' , it means that there is a path c in G which crosses e_1 or e_2 of the same length.
- If H is the result of the operation S_f on a pair of edges e_1, e_2 of weight i and j incident to a vertex v of degree 2, into an edge e' of weight $i + j$, then:

- If $v \notin c$, then the path c is the same in H ;
- If $v \in c$, then c crosses e_1 and e_2 (since v has degree 2 and is neither s nor t). In this case, there is a path c' in H that is the same as c but that crosses e' instead of e_1 and e_2 . The length of this path is $d - l(e_1) - l(e_2) + l(e') = d - i - j + (i + j) = d$. Conversely, a path c' that crosses e' in H implies the existence of a path c in G which crosses the contracted vertex v . These paths have the same length. ◀

Lemma 25 guarantees that the length of the paths from s to t are preserved no matter how many times the operations P_f and S_f are applied. Therefore, if there is a longer path in G , it will be possible to find a path of the same length in H after applying a succession of P_f and S_f operations.

► **Lemma 26.** *Let G be a TTSP graph between s and t , the distance between s and t is robust if and only if G is a TTSP of fixed length.*

Proof. ◀ Lemma 25 shows that if G is turned into K_2 with a succession of operations P_f and S_f , then all paths from s to t in G have length $d(s, t)$, meaning the distance between s and t is robust.

⇒ Let G be a TTSP graph between s and t such that the distance between s and t is robust. Suppose that G can not be reduced to K_2 with a succession of operations P_f and S_f , that is, there is a graph H obtained from G by these operations that cannot be reduced any further and is not K_2 :

- If S_f cannot be applied to H , then H does not have any vertex of degree 2 (except s and t), else it would be possible to sum the weight of the edges with the application of S_f ;
- If P_f cannot be applied to H , then one of the following must hold:
 - H does not have any multiple edges and with S_f impossible, that would mean that G is not TTSP;
 - H has a pair of parallel edges e, f of distinct weights, thus there are in H two paths of different length between u and v , and same in G (by Lemma 25). By Lemma 5, it would mean that the distance is not robust. ◀

Finally the following theorem can be proved:

► **Theorem 27.** *Let G be a connected graph, u, v two vertices of G and H the graph induced by the paths from u to v . The distance between u and v is robust if and only if H is a TTSP of fixed length between u and v .*

Proof. The proof combines several previous results:

- by Lemma 7, the distance is robust in G \iff it is robust in H ;
- by Lemma 23, the distance is robust in H \implies H is TTSP;
- by Lemma 26, if H is TTSP, then the distance is robust \iff H is TTSP of fixed length.

It can be deduced that if the distance between u and v is robust in G , then H is a TTSP of fixed length between u and v . Reciprocally, if H is a TTSP of fixed length between u and v , then the distance between the two vertices is robust in G . ◀

This theorem means that determining the robustness of the distance between two vertices s, t in a graph G can be done efficiently by performing Algorithm 1 (see below). Our algorithm is heavily based on the recognition of TTSP by applying the operations S and P from [16]. Here, instead, we apply the operations P_f and S_f designed for TTSPf from Definition 24. The original algorithm that uses the TTSP operations runs in $O(n + m)$ time. In order to prove that our algorithm runs in linear time, we will describe the main differences from the TTSP-recognition algorithm. Our algorithm performs the following steps:

1. Extract the graph H from G induced by all paths from s to t (by Lemma 7, the robustness of the distance in H is equivalent to the property in G). Extracting H is similar to finding every biconnected component crossed by a path from s to t . Finding the block decomposition is in $O(n + m)$ time [8]. Finding any path from s to t can be done in $O(n + m)$ time as well in an unweighted graph by doing a Breadth-First-Search. The time complexity of this step is $O(n + m)$;
2. If H is unweighted, we initiate a weight of 1 on each edge of H , this is done in $O(m)$;
3. Check if H is a TTSPf by applying the algorithm from [16]. The only operations added are when applying P_f and S_f instead of P and S . Considering that P and S are applied once per edge (as it is done in the TTSP-recognition algorithm), we only need to verify that we add a constant number of operations for each use of P_f and S_f . First of all, with S_f , applied in Lines 22-33, in the original algorithm, the vertex v of degree 2 is removed with its two edges e_1, e_2 and a new edge e is added to connect its neighbors v_1, v_2 , creating a potential multiple edge. Here, the newly created edge e has a weight equal to the sum of the deleted edges e_1, e_2 as shown in Line 26, adding two integers is a constant operation performed once per S_f operations. With the P_f operation, instead of checking every edge of the adjacency list, the original algorithm checks the first edge in the adjacency list of v . After removing the invalid edges in Line 12 that were virtually removed in Lines 17 and 28, the algorithm then checks if a pair of edges that share the same endpoints, in which case it applies P . Here, we first make sure that both edges have the same weight as shown in Line 15, if their weight is different, then the graph is not a TTSPf since P_f would not be applicable. The same kind of verification is done near the end of the algorithm between the remaining edges between s and t in Line 40. Since this verification is only a comparison of integers, it can be done in constant time, as stated before, P_f and S_f being done once per edge, this step of our algorithm adds a complexity of $O(m)$ to the complexity of the original algorithm, therefore this step runs in $O(n + m)$.

5 Concluding remarks and open questions

In this paper, we have shown that the concept of a robust diameter is quite different from the one of robust pairwise distances, so much so that the corresponding decision problems have very different complexities. In the case of the distance, we have identified and exploited a strong connection between TTSP graphs and robust distances, which allowed us to design a linear time algorithm for testing if a given distance is robust. It would be interesting to consider more relaxed versions of the robustness, where one does not ask only whether the distance (or diameter) remains exactly the same, but also whether the deterioration may preserve some comparative quality (this information would have a more practical use). For example, how difficult is it to decide if the distance between u and v may deteriorate up to $d(u, v) + k$ for a fixed k ? Similarly, can the robustness of diameter be approximated in the sense of deciding whether the diameter may deteriorate beyond a certain factor of its original value? Beyond the particular case of robust distances and diameter, the study of robust properties in general is in its infancy, and it would be interesting to see if some meta-theorems can be obtained for robust properties in general.

■ **Algorithm 1** Determination of the robustness of the distance.

Data: $G = (V, E), s, t \in V(G)$
Result: *True* iff $dist(s, t)$ is robust

```

1  $H \leftarrow inducedPathsGraph(G, s, t);$ 
2  $(order, size) \leftarrow (|V(H)|, |E(H)|);$ 
3 for  $e \in E(H)$  do
4    $e.weight \leftarrow 1;$ 
5    $e.valid \leftarrow True;$                                /* Presence of edges in H */
6  $X \leftarrow V(H) \setminus \{s, t\};$ 
7 while  $X \neq \emptyset$  do
8    $v \leftarrow X.removefirst();$ 
9   while  $v.degree() > 2$  do
10     $(e_1, e_2, e_3) \leftarrow (v.edges()[0], v.edges()[1], v.edges()[2]);$ 
11    if  $\exists e \in \{e_1, e_2, e_3\}, e.valid = False$  then
12       $v.edges().delete(e);$ 
13    else if  $\exists (e, f) \in \{e_1, e_2, e_3\}, e, f$  have the same endpoints then
14      if  $e.weight \neq f.weight$  then
15         $\text{return } False;$                                /*  $P_f$  cannot be applied */
16      else
17         $f.valid \leftarrow False;$ 
18         $size \leftarrow size - 1;$ 
19         $v.edges().delete(f);$ 
20    else
21       $\text{exit while};$ 
22  if  $v.degree() = 2$  then
23     $(e_1, e_2) \leftarrow v.edges();$                        /* Application of  $S_f$  */
24     $(v_1, v_2) \leftarrow v.neighbors();$ 
25     $e \leftarrow NewEdge(v_1, v_2);$ 
26     $e.weight \leftarrow e_1.weight + e_2.weight;$ 
27     $H.addEdge(e);$ 
28     $(e_1.valid, e_2.valid) \leftarrow (False, False);$ 
29     $(order, size) \leftarrow (order - 1, size - 1);$ 
30    if  $v_1 \neq s$  and  $v_1 \neq t$  then
31       $X.add(v_1);$ 
32    if  $v_2 \neq s$  and  $v_2 \neq t$  then
33       $X.add(v_2);$ 
34  if  $order > 2$  then
35     $\text{return } False;$ 
36  if  $size > 1$  then
37     $w \leftarrow E(H)[0].weight;$                        /* Final application of  $P_f$  on  $st$  edges */
38    for  $e \in E(H)$  do
39      if  $e.weight \neq w$  then
40         $\text{return } False;$ 
41   $\text{return } True;$ 

```

References

- 1 Ingo Althöfer, Gautam Das, David Dobkin, Deborah Joseph, and José Soares. On sparse spanners of weighted graphs. *Discrete & Computational Geometry*, 9(1):81–100, 1993.
- 2 Eli Berger, Paul Seymour, and Sophie Spirkl. Finding an induced path that is not a shortest path. *Discrete Mathematics*, 344(7):112398, 2021.
- 3 Arnaud Casteigts, Swan Dubois, Franck Petit, and John M Robson. Robustness: A new form of heredity motivated by dynamic networks. *Theoretical Computer Science*, 806:429–445, 2020.
- 4 R.J Duffin. Topology of series-parallel networks. *Journal of Mathematical Analysis and Applications*, 10(2):303–318, 1965.
- 5 David Eppstein. Parallel recognition of series-parallel graphs. *Information and Computation*, 98(1):41–55, 1992.
- 6 Cyril Gavoille, Quentin Godfroy, and Laurent Viennot. Node-disjoint multipath spanners and their relationship with fault-tolerant spanners. In *International Conference On Principles Of Distributed Systems*, pages 143–158, 2011.
- 7 Frank Harary. *Graph Theory*. Addison-Wesley Publishing Company, 1st edition, 1969.
- 8 John Hopcroft and Robert Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.
- 9 Kuo-Hua Kao, Jou-Ming Chang, Yue-Li Wang, and Justie Su-Tzu Juan. A quadratic algorithm for finding next-to-shortest paths in graphs. *Algorithmica*, 61(2):402–418, 2011.
- 10 Ilia Krasikov and Steven D Noble. Finding next-to-shortest paths in a graph. *Information processing letters*, 92(3):117–119, 2004.
- 11 Kumar N Lalgudi and Marios C Papaefthymiou. Computing strictly-second shortest paths. *Information processing letters*, 63(4):177–181, 1997.
- 12 Shisheng Li, Guangzhong Sun, and Guoliang Chen. Improved algorithm for finding next-to-shortest paths. *Information processing letters*, 99(5):192–194, 2006.
- 13 Karl Menger. Zur allgemeinen kurventheorie. *Fundamenta Mathematicae*, 10(1):96–115, 1927.
- 14 John Riordan and Claude E Shannon. The number of two-terminal series-parallel networks. *Journal of Mathematics and Physics*, 21(1-4):83–93, 1942.
- 15 Neil Robertson and Paul D Seymour. Graph minors. xiii. the disjoint paths problem. *Journal of combinatorial theory, Series B*, 63(1):65–110, 1995.
- 16 Jacobo Valdes, Robert E Tarjan, and Eugene L Lawler. The recognition of series parallel digraphs. In *Proc. of the 11th ACM Symposium on Theory of Computing*, pages 1–12, 1979.
- 17 Paul Wollan. *Extremal functions for graph linkages and rooted minors*. Georgia Institute of Technology, 2005.

Computing Outside the Box: Average Consensus over Dynamic Networks

Bernadette Charron-Bost ✉

Département d'informatique de l'ENS, ENS, CNRS, PSL University, Paris, France

Patrick Lambein-Monette¹ ✉ 

Université Paris Cité, CNRS, IRIF, F-75013, Paris, France

Abstract

Networked systems of autonomous agents, and applications thereof, often rely on the control primitive of *average consensus*, where the agents are to compute the average of private initial values. To provide reliable services that are easy to deploy, average consensus should continue to operate when the network is subject to frequent and unpredictable change, and should mobilize few computational resources, so that deterministic, low powered, and anonymous agents can partake in the network.

In this stringent adversarial context, we investigate the implementation of average consensus by distributed algorithms over networks with bidirectional, but potentially short-lived, communication links. Inspired by convex recurrence rules for multi-agent systems, and the Metropolis average consensus rule in particular, we design a deterministic distributed algorithm that achieves asymptotic average consensus, which we show to operate in polynomial time in a synchronous temporal model.

The algorithm is easy to implement, has low space and computational complexity, and is fully distributed, requiring neither symmetry-breaking devices like unique identifiers, nor global control or knowledge of the network. In the fully decentralized model that we adopt, to our knowledge, no other distributed average consensus algorithm has a better temporal complexity.

Our approach distinguishes itself from classical convex recurrence rules in that the agent's values may sometimes leave their previous convex hull. As a consequence, our convergence bound requires a subtle analysis, despite the syntactic simplicity of our algorithm.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Computing methodologies → Distributed artificial intelligence; Networks → Sensor networks; Networks → Mobile networks; Networks → Network dynamics

Keywords and phrases average consensus, dynamic networks, distributed algorithms, iterated averaging, Metropolis

Digital Object Identifier 10.4230/LIPIcs.SAND.2022.10

Related Version *Extended Version*: <https://arxiv.org/abs/2010.05675>

Acknowledgements Patrick Lambein-Monette would like to thank his doctoral jury for stimulating discussions and remarks regarding previous versions of this material.

1 Introduction

1.1 Asymptotic average consensus

We consider a networked system of n *agents* – the generic term we use to denote the autonomous nodes of the network – denoted by the integer labels $1, \dots, n$. Agent i begins with an *input* value $\mu_i \in \mathbb{R}$, and maintains an *estimate* $x_i(t)$ of an objective. The input represents the agent's private observation of some aspect of its environment, which we assume to be taken arbitrarily from the domain of the problem; for example, the input may be a temperature reading, or the agent's initial position in space or velocity, if it is mobile. The

¹ Corresponding author



estimate represents some aspect of the environment affected by the agent; depending on the system, it may simply be a local variable in the agent's memory, or it may directly represent some external parameter like the agent's heading or altitude.

Here, we focus on (asymptotic) *average consensus*, a control primitive widely studied by the distributed control community, where the estimates are made to *achieve asymptotic consensus* on the average of the input values – that is, to jointly converge towards the same limit $\bar{\mu} := \frac{1}{n} \sum_i \mu_i$. The problem of computing an average is central to many applications in distributed control: let us cite sensor fusion and data aggregation [37, 27, 36], distributed optimization and machine learning [24, 28, 26], collective motion [32, 30], and more [13, 8, 12]. More generally, an average consensus primitive can be used to compute the relative frequency of the input values [16], and as such allows for the distributed computation of other statistical measures, for example the *mode* – the value with the highest support.

We study the problem of designing distributed algorithms for average consensus in the adversarial context of *dynamic networks*, where the communication links joining the agents change over time. Indeed, average consensus primitives are often needed in inherently dynamic settings, that static models fail to adequately describe. For a few examples, let us cite mobile ad-hoc networks, where links change as external factors cause the agents to move in space; autonomous vehicular networks, where agents are in control of their motion; or peer-to-peer networks, where constant arrivals and departures cause the network to reconfigure.

Specifically, we study distributed algorithms in a fully decentralized context: all agents start in the same state, run the same local algorithm, receive no global information about the system, only manipulate local variables, and interact with the system exclusively by exchanging messages with neighboring agents in the instantaneous communication graph. These constraints preclude the use of many standard solutions where the agents receive unique identifiers, where an agent is designated as a leader, or where all agents initially agree on a bound on the network's degree or size. Moreover, we adopt a standard *local broadcast* communication model, particularly suited to modeling wireless networks, in which agents cast their messages without knowledge of their eventual recipients, and in particular cannot individually address their neighbors.

These conditions make it extremely hard to compute functions of the input values μ_1, \dots, μ_n : on general *fixed* directed networks, deterministic distributed algorithms are only capable of computing functions that depend on the *set* of the input values $\{\mu_1, \dots, \mu_n\}$, but not on their *multi-set* [17]. In particular, this precludes the distributed computation of the average. Here, we only consider networks with bidirectional communication links. Under this condition, the problem is rather simple if we assume a static communication graph [37, 5], in which case we can even deploy efficient solutions [31, 28] relying on spectral properties of the underlying graph. The problem is obviously much harder in a dynamic setting, which, for example, forbids the use of such sophisticated spectral techniques.

1.2 Contribution

A standard approach to asymptotic consensus has agents regularly adjust their estimates as a convex combination of those of their neighbors [10, 33], defined by a *convex recurrence rule*. We adopt a standard model of *synchronized rounds*, where this is expressed as a recurrence relation taking the generic form $x_i(t) = \sum_{j \in \mathcal{N}_i(t)} a_{ij}(t) x_j(t-1)$, where the weights $a_{ij}(t)$ are taken to form a convex combination, and the sum is over an agent's incoming neighbors in the communication graph at round t .

While asymptotic consensus is guaranteed as long as the never permanently splits [22], the estimates do not, in general, converge towards the average $\bar{\mu}$; reaching *average* consensus usually requires additionally enforcing *symmetric* weights $a_{ij}(t) = a_{ji}(t)$. Here, we study *distributed algorithms* for average consensus, i.e., we are interested in devising an algorithm that produces such weights through local computations only, in a fully decentralized manner.

For a simple example, average consensus comes easily by picking the weights $a_{ij}(t) = \frac{1}{n}$ when agents $i \neq j$ are neighbors in round t , and $a_{ii}(t) = 1 - \frac{\deg_i(t)-1}{n}$. However, this scheme might be simple to *describe*, but getting the agents to use these weights clearly requires getting them to know n , which is itself a serious distributed computing problem.

We will argue that the Metropolis rule [37], defined by the weights $a_{ij}(t) = \frac{1}{\max(\deg_i(t), \deg_j(t))}$ for any two $i \neq j$ neighbors in round t , breaks down over dynamic networks because of similar, albeit subtler, issues. We then propose a symmetric recurrence rule that *is* implementable over dynamic bidirectional networks, that we show to produce average consensus over any sufficiently connected network. The issues faced by the Metropolis rule are overcome by making the rule sometimes break convexity, which allows for keeping the average of the estimates constant even though the network changes unpredictably.

The temporal convexity of our distributed algorithm is polynomial, namely with a bound in $O(n^4 \log n)$, whereas the *theoretical* complexity bound of the Metropolis rule is of $O(n^2 \log n)$ [5]. To the best of our knowledge, this is the first deterministic algorithm that achieves asymptotic average consensus over bidirectional dynamic networks without any centralized input or symmetry-breaking assumptions. We note in passing that there exist *randomized* algorithms that are efficient in bandwidth and memory and converge in $O(n)$ rounds to a good approximation of the average $\bar{\mu}$ with high probability [6, 20, 23].

We dub our distributed algorithm *MaxMetropolis*. Compared to the Metropolis rule, the change that we propose is deceptively simple: in the expression of the Metropolis weights, we replace the degree $\deg_i(t)$ with the value $\overline{\deg}_i(t-1) = \max\{\deg_i(1), \dots, \deg_i(t-1)\}$. However, the resulting rule is no longer *convex* – the estimates $x_i(t)$ may sometimes leave the convex hull of the set $\{x_1(t-1), \dots, x_n(t-1)\}$ – which makes the analysis substantially harder than in the purely convex case. Interestingly, although such “bad”, convexity-breaking rounds, can happen at an arbitrarily late stage in the execution, we are able to bound the convergence time *independently* of when bad rounds occur – that is, once our target error threshold has been reached, disagreement in the system can still increase in later bad rounds, but not enough to break the threshold again.

1.3 Related works

Average consensus itself is at the center of a large body of works: among many others, let us cite [33, 34, 8, 19, 35, 37, 25, 3, 13, 28, 14], and see [26] for a recent overview of the domain. The approach based on doubly stochastic matrices in particular has been studied in depth, notably in [25, 29], with an analytical approach that focuses on aspects such as the temporal complexity and tolerance to quantization, whereas we address issues of a distributed nature, in particular the implementation of rules by distributed algorithms. We also note earlier work on random walks by Avin et al., who showed that dynamic networks can present considerable obstacles to mixing, in stark contrast with the well-behaved static case. Although their proposed solution is not directly implementable in our model, as it leverages global information (a bound over n), their study nonetheless deeply influenced the current work.

Of interest to our argument, we note that [35] looks for the *fixed* affine weights that optimize the speed of convergence towards average consensus over a given fixed graph, and find that the weights can often be negative. Our algorithm is itself able to solve average

consensus over dynamic networks precisely because it is sometimes allowed to use negative weights. When compared with our approach, the important difference is that we consider dynamic graphs and focus on distributed implementation of the recurrence rules, while the weights obtained in [35] are given by a centralized optimization problem, and are incompatible with a distributed approach.

A number of strategies aim at speeding up convex recurrence rules over static networks by having the agents learn what amounts to spectral elements of the graph Laplacian [4], and can result in linear-time convergence [31]. As is the case here, these represent distributed methods by which the agents learn structural properties of the communication graph. However, these methods rely on centralized symmetry-breaking crutches like unique identifiers, and their memory and computation footprint is much greater than ours, with agents computing and memorizing, in each round, the kernels of Hankel matrices of dimension $\Theta(n) \times \Theta(n)$. In contrast, our method can be used by anonymous agents, requires $\lceil \log n \rceil$ additional bits of memory and bandwidth, and has a trivial computational overhead.

2 Preliminaries

2.1 Mathematical toolbox

Let us fix some notation. If k is a positive integer, we denote by $[k]$ the set $\{1, \dots, k\}$. If any set $S \subset \mathbb{R}$ is non-empty and bounded, we denote its *diameter* by $\text{diam } S := \max S - \min S$.

A *directed graph* (or simply *graph*) $G = (V, E)$, with vertices in V and edges in $E \subseteq V \times V$, is called *reflexive* when $(i, i) \in E$ for all $i \in V$; G is *bidirectional* when $(i, j) \in E \iff (j, i) \in E$ for all $i, j \in V$; and G is *strongly connected* when directed paths join any pair of vertices – or simply *connected* when G is bidirectional.

All graphs that we consider here will be reflexive, bidirectional, and connected graphs of the form $G = ([n], E)$. In such a graph, the vertices linked to some vertex i form its *neighborhood* $\mathcal{N}_i(G) := \{j \in [n] \mid (j, i) \in E\}$, and the count of its neighbors is its *degree* $\text{deg}_i(G) := |\mathcal{N}_i(G)|$. By definition, the degree is at most n , and in a reflexive graph it is at least 1.

We consistently denote matrices and vectors in bold italic style: upper case for matrices (e.g., \mathbf{A}) and lower case for vectors (e.g., \mathbf{u}), with their individual entries in regular italic style, (e.g., A_{ij}, u_k). The shorthand $\mathbf{v}^{\mathbb{N}}$ denotes the infinite vector sequence $\mathbf{v}(0), \mathbf{v}(1), \dots$.

The graph $G_{\mathbf{A}} = ([n], E)$ associated to a matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ is defined by $(j, i) \in E \iff A_{ij} \neq 0$ for all $i, j \in [n]$. The matrix \mathbf{A} is said to be *irreducible* when $G_{\mathbf{A}}$ is strongly connected.

Given a vector $\mathbf{v} \in \mathbb{R}^n$, we write $\text{diam } \mathbf{v}$ to mean the diameter of the set $\{v_1, \dots, v_n\}$ of its entries. The diameter constitutes a seminorm over \mathbb{R}^n ; we call *consensus vectors* those of null diameter.

A matrix or a vector with non-negative (resp. positive) entries is itself called *non-negative* (resp. positive). A vector is called *stochastic* if its entries are non-negative sum to 1.

A matrix \mathbf{A} is *stochastic* if its rows are all *stochastic* – that is, if $\mathbf{A}\mathbf{1} = \mathbf{1}$ – and any matrix that satisfies the condition $\mathbf{A}\mathbf{1} = \mathbf{1}$ will be said to be *affine*. We say that a matrix \mathbf{A} is *doubly stochastic* when both \mathbf{A} and \mathbf{A}^{\top} are stochastic.

We denote the mean value of a vector $\mathbf{v} \in \mathbb{R}^n$ by $\langle \mathbf{v} \rangle := \frac{1}{n} \sum_i v_i$. Doubly stochastic matrices play a central role in the study of average consensus, as multiplying any vector \mathbf{v} by a doubly stochastic matrix \mathbf{A} preserves its average – that is, $\langle \mathbf{A}\mathbf{v} \rangle = \langle \mathbf{v} \rangle$.

For any matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, we can arrange its n eigenvalues $\lambda_1, \dots, \lambda_n$, counted with their algebraic multiplicities, in decreasing order of magnitude: $|\lambda_1| \geq |\lambda_2| \geq \dots \geq |\lambda_n|$.

Under this convention, the *spectral radius* of the matrix A is the quantity $\rho_A := |\lambda_1|$, and its *spectral gap* is the quantity $\gamma_A := |\lambda_1| - |\lambda_2|$. In particular, a stochastic matrix has a spectral radius of 1, which is itself an eigenvalue for the eigenvector $\mathbf{1}$.

2.2 Computing model

We consider a networked system of n agents, denoted $1, 2, \dots, n$. Computation proceeds in *synchronized rounds* that are communication closed, in the sense that no agent receives messages in round t that are sent in a different round. In each round $t \in \mathbb{N}_{>0}$, each agent i successively

1. broadcasts a *single* message $m_i(t)$ determined by its state at the beginning of round t
2. receives *some* messages among $m_1(t), \dots, m_n(t)$
3. undergoes an internal transition to a new state
4. produces a *round output* $x_i(t) \in \mathbb{R}$ and proceeds to round $t + 1$.

The agents receiving agent i 's message $m_i(t)$ are unknown to agent i at the time of emission, in step 1. Communications that occur in round t are modeled by a directed graph $\mathbb{G}(t) := ([n], E(t))$, called the round t *communication graph*, which may change from one round to the next. We assume each communication graph $\mathbb{G}(t)$ to be reflexive, as an agent always has access to its own messages without delay or transmission loss.

Messages to be sent in step 1 and state transitions in step 3 are determined by a *sending* and a *transition* functions, which together define the *local algorithm* for agent i . Collected together, the local algorithms of all agents in the system constitute a *distributed algorithm*. We posit no *a priori* global coordination or knowledge of the agents: in particular, we assume no leader, no unique identifiers, and no initial agreement on global parameters such as n . An agent's computations only involve its own local variables in memory.

An *execution* of a distributed algorithm is a sequence of rounds, as defined above, with each agent running the corresponding local algorithm. We assume that all agents start simultaneously in round 1, since the algorithms under our consideration are robust to asynchronous starts, retaining the same time complexity as when the agents start simultaneously. Indeed, asynchronous starts only induce an initial transient period during which the network is disconnected, which cannot affect the convergence and complexity results of algorithms driven by convex recurrence rules.

In any execution of a distributed algorithm, the entire sequence $\mathbf{x}^{\mathbb{N}}$ is determined by the input vector $\boldsymbol{\mu}$ and the patterns of communications in each round t , i.e., the sequence of communication graphs $\mathbb{G} := (\mathbb{G}(t))_{t \geq 1}$, called the *dynamic communication graph* of the execution, and so we write $\mathbf{x}^{\mathbb{N}} = \mathbf{x}^{\mathbb{N}}(\mathbb{G}, \boldsymbol{\mu})$. When the dynamic graph \mathbb{G} is understood, we let $\mathcal{N}_i(t)$ and $\deg_i(t)$ respectively stand for $\mathcal{N}_i(\mathbb{G}(t))$ and $\deg_i(\mathbb{G}(t))$. As no confusion can arise, we will sometimes identify an agent with its corresponding vertex in the communication graph, and speak of the degree or neighborhood of an *agent* in a round of an execution.

We call a *network class* a set of dynamic graphs; given a class \mathfrak{C} , we denote by $\mathfrak{C}|_n$ the subclass $\{\mathbb{G} \in \mathfrak{C} \mid |\mathbb{G}| = n\}$. Here, our investigation will revolve around the class $\mathfrak{B}_{\mathfrak{C}}$ of dynamic graphs of the following sort.

► **Assumption 1.** *In each round $t \in \mathbb{N}_{>0}$, the communication graph $\mathbb{G}(t)$ is reflexive, bidirectional, and connected.*

3 Recurrence rules for consensus

We distinguish local algorithms, as defined above, from the *recurrence rules* that they implement: the latter are recurrence relations that only describe how the estimates $x_i(t)$ change over time, while the former specifies the *distributed implementation* of such rules in the system, through local interactions. This discrepancy is apparent in the Metropolis rule, whose distributed implementation over dynamic networks is problematic due to its dependence on “knowledge at distance two”.

3.1 Affine recurrence rules

Definition

Here, we focus on algorithmic solutions to the average consensus problem whose executions realize recurrence relations of the general form

$$x_i(t) = \sum_{j \in \mathcal{N}_i(t)} a_{ij}(t) x_j(t-1), \quad (1)$$

where the time-varying weights $a_{ij}(t)$ satisfy the affine constraint $\sum_{j \in \mathcal{N}_i(t)} a_{ij}(t) = 1$ and may depend on the dynamic graph \mathbb{G} and the input values μ_1, \dots, μ_n . We refer to such relations as *affine recurrence rules*, and we say that a distributed algorithm *implements* the rule, insisting again that *a distributed algorithm is distinct from the rule it implements*.

Because of the constraint $\sum_{j \in \mathcal{N}_i(t)} a_{ij}(t) = 1$, the self-weights satisfy $a_{ii}(t) = 1 - \sum_{j \in \mathcal{N}_i(t) \setminus \{i\}} a_{ij}(t)$. An affine recurrence rule is thus fully specified by the weights $a_{ij}(t)$ assigned to an agent’s proper neighbors $j \neq i$.

The affine rule of Equation (1) is equivalent to the vector equation $\mathbf{x}(t) = \mathbf{A}(t)\mathbf{x}(t-1)$, where $A_{ij}(t) = a_{ij}(t)$ when i and j are neighbors in round t , and $A_{ij}(t) = 0$ otherwise. The affinity constraint then corresponds to the condition $\mathbf{A}(t)\mathbf{1} = \mathbf{1}$.

Convexity and convergence

We call the rule *convex* when all weights are non-negative – equivalently, when all matrices $\mathbf{A}(t)$ are stochastic. By and large, the study of affine recurrence rules focuses on that of convex recurrence rules, which guarantee convergence under mild conditions. We recall a standard convergence result, found under various forms in the literature, see for example [7, 33, 18, 22].

► **Proposition 2.** *Assume that the weights of Equation (1) admit a uniform positive lower bound α : $a_{ij}(t) \geq \alpha > 0$ for all t, i , and $j \in \mathcal{N}_i(t)$. Under Assumption 1, the vectors $\mathbf{x}(t)$ converge to a consensus vector.*

We speak of *uniform convexity* when such a parameter α exists, and we note that in this case asymptotic consensus is actually ensured by conditions much weaker than Assumption 1: for bidirectional interactions, it is enough that the network never become permanently split [22, Theorem 1].

Remark that Proposition 2 says nothing of the *value* of the consensus; affine recurrence rules for *average* consensus are typically designed to produce matrices that are doubly stochastic. By enforcing the invariant $\langle \mathbf{x}(t) \rangle = \langle \mathbf{x}(t-1) \rangle$, this makes the initial average $\bar{\mu}$ the only admissible consensus value.

The *convergence time* of a single sequence $\mathbf{z}^{\mathbb{N}}$, given by $T(\varepsilon; \mathbf{z}^{\mathbb{N}}) := \inf\{t \in \mathbb{N} \mid \forall \tau \geq t: \text{diam } \mathbf{z}(\tau) \leq \varepsilon\}$, measure its progress towards asymptotic consensus. For a rule or an algorithm, we consider the more helpful *worst-case relative convergence time* over a class \mathfrak{C} : for a system of n agents, it is defined by

$$T(\varepsilon; n, \mathfrak{C}) := \sup_{\boldsymbol{\mu} \in \mathbb{R}^n} \sup_{\mathbb{G} \in \mathfrak{C}_{|n}} T(\varepsilon \cdot \text{diam } \boldsymbol{\mu}; \mathbf{x}^{\mathbb{N}}(\mathbb{G}, \boldsymbol{\mu})), \quad (2)$$

where we drop the class \mathfrak{C} if it is clear from the context.

We recall the following bounds for uniformly convex recurrence rules over the class $\mathfrak{B}_{\mathfrak{C}}$: when all matrices are *doubly stochastic*, the convergence time is in $O(\alpha^{-1}n^2 \log n/\varepsilon)$ [25, Theorem 10]. In the common case that $\alpha = \Theta(1/n)$, all rules are known to admit executions that do not converge before $\Omega(n^2 \log 1/\varepsilon)$ rounds over the fixed line graph with n vertices [29, Theorem 6.1].

3.2 Consensus and average consensus rules

The EqualNeighbor rule

The prototypical example of a convex recurrence rule is the *EqualNeighbor* rule, where an agent assigns the equal weights to all its neighbors, itself included:

$$x_i(t) = \frac{1}{\text{deg}_i(t)} \sum_{j \in \mathcal{N}_i(t)} x_j(t). \quad (3)$$

We can mechanically derive an algorithm implementing the EqualNeighbor rule: in each round t , broadcast one's latest estimate $x_i(t-1)$, and pick as new estimate $x_i(t)$ the arithmetic mean of the incoming values. Since $\text{deg}_i(t) \leq n$, this rule admits $1/n$ as a parameter of uniform convexity, and for a dynamic graph of $\mathfrak{B}_{\mathfrak{C}}$, Proposition 2 shows that any solution to Equation (3) converges to a consensus vector.

Clearly, the EqualNeighbor rule does *not* solve the *average* consensus problem on the entire class $\mathfrak{B}_{\mathfrak{C}}$, as the weights are generally not symmetric, unless each communication graph $\mathbb{G}(t)$ is *regular* – that is, if all its vertices have the same degree.

The Metropolis rule

In [37], Xiao et al. investigate the problem of *distributed sensor fusion* with the help of an average consensus primitive. For that, they describe the “maximum-degree” rule, parametrized with an integer $N \geq 1$, defined by the constant weights $a_{ij}(t) = 1/N$ for any agents $i \neq j$ neighbors in round t .

The authors note that this rule solves average consensus over the class $\cup_{n \leq N} \mathfrak{B}_{\mathfrak{C}|n}$, but remark that *implementing* this rule hinges on the agents initially agreeing on the bound N , embedding an assumption of centralized control. This makes the “maximum-degree” rule inapplicable over truly decentralized systems – indeed, our communication model does not generally allow for the distributed computation of such a bound N [1]. Xiao et al. go on suggesting the alternative rule:

$$x_i(t) = x_i(t-1) + \sum_{j \in \mathcal{N}_i(t)} \frac{x_j(t-1) - x_i(t-1)}{\max(\text{deg}_i(t), \text{deg}_j(t))}, \quad (4)$$

generally referred to as the *Metropolis* rule, as it is inspired from the Metropolis-Hastings method [15, 21].

Analytically, this rule is appealing, as it was recently shown [5] to display a worst-case convergence time of $O(n^2 \log n)$ over the entire class \mathfrak{B}_c – making it the fastest rule known to us to solve either consensus or average consensus on that class. From a computational perspective, it is argued in [37] that the Metropolis rule is better suited for decentralized systems, as it only leverages “local” knowledge. Indeed, agents can implement this rule knowing only, in each round, their own degrees in the current communication graph and that of their neighbors – compared to the initial agreement over $N \geq n$ required of the “maximum-degree” rule.

Unfortunately, local algorithms cannot implement the Metropolis rule over dynamic networks. The rule is only “local” in the weak sense that an agent’s next estimate $x_i(t)$ depends on information present *within distance* 2 of agent i in the communication graph $\mathbb{G}(t)$, which is not local *enough* when the network is subject to change.

Indeed, since agent $j \in \mathcal{N}_i(t)$ only learns its round t degree $\deg_j(t)$ at the *end* of round t – by counting its incoming messages – it cannot share this information with other agents before the following round. Any distributed implementation of the Metropolis rule would therefore require communication links that evolve at a slow and regular pace; one can imagine a network whose topology can only change once every k rounds, when $t \equiv 0 \pmod k$, e.g., at even rounds.

When the network is subject to *unpredictable* changes, the situation is even worse: we need to warn all agents, ahead of time, about any upcoming topology change. In effect, this amounts to having a global synchronization signal precede every change in the communication topology. For a topology change in round t_0 , this differs little from starting an *entirely new execution* with new input values $\mu'_1 = x_1(t_0 - 1), \dots, \mu'_n = x_n(t_0 - 1)$. To paraphrase, given a sufficiently stable communication network, one “can” implement the Metropolis rule over dynamic networks; however, the execution is fully decentralized only as long as no topology change actually occurs.

We note that, although we have covered the Metropolis rule here, other average consensus rules typically face similar problems, even when expressly designed for dynamic networks. As an example, while the Metropolis rule can be implemented with a two-message protocol – e.g., on a communication graph that changes every other round, and with all agents agreeing on the parity of the round number, see e.g., [9] for a discussion – the rules given in [29, Algorithm 8.2] and [25, Section IV.A] involve a three-message protocol. Their implementation thus requires more network stability, and a stronger agreement, than Metropolis.

4 The MaxMetropolis algorithm

4.1 A symmetric affine rule

Symmetrizing

Let us briefly recall the idea of the Metropolis-Hastings [15, 21] method: given a positive stochastic vector $\boldsymbol{\pi}$, the method turns a stochastic matrix \mathbf{A} – usually viewed as the transition matrix of a reversible Markov chain – into another stochastic matrix \mathbf{A}' with stationary distribution $\boldsymbol{\pi}$, by picking off-diagonal entries as $A'_{ij} = \min\left(A_{ij}, \frac{\pi_j}{\pi_i} A_{ji}\right)$. When $\boldsymbol{\pi}$ is the constant vector $(\frac{1}{n}, \frac{1}{n}, \dots, \frac{1}{n})$, we get the simpler transform $M(-)$, defined entry-wise by:

$$\forall i, j \in [n]: [M(\mathbf{A})]_{ij} = \begin{cases} \min(A_{ij}, A_{ji}) & j \neq i \\ 1 - \sum_{k \neq i} \min(A_{ik}, A_{ki}) & j = i. \end{cases} \quad (5)$$

Let us call this transform the *Metropolis-Hastings symmetrization*; as an example, the symmetrization of the EqualNeighbor matrix yields the Metropolis matrix. We can make a few remarks: for any matrix \mathbf{A} , the matrix $M(\mathbf{A})$ is affine and symmetric by construction, and for any $j \neq i$ we have $[M(\mathbf{A})]_{ij} \leq A_{ij}$ and therefore $[M(\mathbf{A})]_{ii} \geq A_{ii}$. In particular, if the matrix \mathbf{A} is stochastic with positive diagonal entries, then so is $M(\mathbf{A})$; if we can use Proposition 2 to establish the convergence of the system $\mathbf{x}(t) = \mathbf{A}(t)\mathbf{x}(t-1)$, then necessarily the system $\mathbf{y}(t) = M(\mathbf{A}(t))\mathbf{y}(t-1)$ also converges, and achieves average consensus.

Bound learning

To apply the Metropolis-Hastings symmetrization while avoiding the aforementioned limitations of the Metropolis rule, let us temporarily assume that each agent $i \in [n]$ initially knows an upper bound $q_i \geq 1$ over its degree throughout the execution, i.e., $q_i \geq \deg_i(t)$ for all $t \geq 1$.

In this case, an agent may broadcast in each round the pair $\langle q_i, x_i(t-1) \rangle$ to its neighbors, and adjust its estimate as

$$x_i(t) = x_i(t-1) + \sum_{j \in \mathcal{N}_i(t)} \frac{x_j(t-1) - x_i(t-1)}{\max(q_j, q_i)}; \quad (6)$$

we easily see that this rule produces symmetric weights ($a_{ij}(t) = a_{ji}(t)$) and has a uniform convexity parameter of $1/\max_i q_i$. For a dynamic graph of \mathfrak{B}_c , any solution $\mathbf{z}^{\mathbb{N}}$ of Equation (6) converges to a consensus vector, by Proposition 2, and therefore achieves asymptotic average consensus, since the weights are symmetric. Using e.g., the aforementioned result of [25, Theorem 10], we can show that the convergence time behaves as $O(\max_i q_i \cdot n^2 \log n/\varepsilon)$, which is polynomial in n when the bounds q_i themselves are.

Obviously, assuming such bounds q_i supposes that the agents dispose of information about the dynamic structure of the network ahead of the execution, which our model explicitly disallows. Instead of *assuming* such bounds, we next show that we can solve the average consensus problem for the class \mathfrak{B}_c by making agents learn good bounds over time in a manner consistent with our symmetric and local model.

To this effect, for each agent i we let $\overline{\deg}_i(t) := \max\{\deg_i(1), \dots, \deg_i(t)\}$ for any round t . For a dynamic graph in $\mathfrak{B}_{c|n}$, the value $\overline{\deg}_i(t) \in [2, n]$ is weakly increasing with t , and therefore stabilizing: we have $\overline{\deg}_i(t) = \overline{\deg}_i := \max_{\tau \geq 1} \deg_i \tau$ for all rounds t beyond some round t_i^* . Thus, by keeping track of $\overline{\deg}_i(t)$, agent i will eventually hold a bound on its *future* degrees for the rest of the execution, which may be used to implement Equation (6), not for the whole interval $[1, \infty[$, but on all but finitely many rounds.

Moreover, we have by definition $\overline{\deg}_i(t) \geq \deg_i(t)$, so that using $\overline{\deg}_i(t)$ in place of q_i in Equation (6) produces a convex rule – even though $\overline{\deg}_i(t)$ may be inferior to agent i 's *future* degree. Unfortunately, the weights $\frac{1}{\max(\overline{\deg}_i(t), \overline{\deg}_j(t))}$ cannot be computed in a local manner: since $\overline{\deg}_i(t)$ depends on $\deg_i(t)$, the issues of the Metropolis rule apply here as well, as an agent cannot communicate its degree to its neighbors at the time they need the information.

We overcome this obstacle with a small, but crucial adjustment: building the round t weights using the *latest known bound* $\overline{\deg}_i(t-1)$ in place of $\overline{\deg}_i(t)$ allows us to conform to the stringent *locality* constraints by sacrificing the *convexity* of the rule. Specifically, we propose the *MaxMetropolis* algorithm – given in Algorithm 1, – a deterministic distributed algorithm which solves the average consensus problem over the class \mathfrak{B}_c in polynomial time, by implementing the rule

$$x_i(t) = x_i(t-1) + \sum_{j \in \mathcal{N}_i(t)} \frac{x_j(t-1) - x_i(t-1)}{\max(\overline{\deg}_j(t-1), \overline{\deg}_i(t-1))}. \quad (7)$$

■ **Algorithm 1** The MaxMetropolis algorithm, code for agent i .

Input: $\mu_i \in \mathbb{R}$

1 **Initially:**

2 $x_i \leftarrow \mu_i$;

3 $q_i \leftarrow 2$;

4 **In each round do:**

5 send $m_i = \langle x_i, q_i \rangle$;

6 receive m_{j_1}, \dots, m_{j_d} ; $\triangleright d$ neighbors

7 $x_i \leftarrow x_i + \sum_{k=1}^d \frac{x_{j_k} - x_i}{\max(q_i, q_{j_k})}$;

8 $q_i \leftarrow \max(q_i, d)$;

9 output x_i ;

The weights are clearly symmetric, and so any solution to Equation (7) satisfies the invariant $\langle \mathbf{x}(t+1) \rangle = \langle \mathbf{x}(t) \rangle$. Moreover, by construction, there exists a round t^* after which we have $\overline{\deg}_i(t-1) = \overline{\deg}_i \geq \deg_i(t)$; the assumptions of Proposition 2 are then satisfied over the infinite interval $[t^*, \infty[$. Taken together, these observations immediately give us that MaxMetropolis is an average consensus distributed algorithm for the class \mathfrak{B}_c .

On the other hand, in contrast with the Metropolis rule, the MaxMetropolis rule offers no guarantee of convexity: we easily see that if, for example, $\deg_i(t)$ is much larger than $\overline{\deg}_i(t-1)$, $x_i(t)$ may leave the convex hull of $\{x_j(t-1) \mid j \in \mathcal{N}_i(t)\}$, and in fact may even leave the convex hull of $\{x_j(t-1) \mid j \in [n]\}$. Such convexity-breaking rounds can occur late in the execution, and our main analytical difficulty will be to show that these “late bad rounds” cannot introduce too much noise in the system once a given degree of agreement has been reached.

► **Theorem 3.** *The MaxMetropolis algorithm solves the average consensus problem in all of its executions over the class \mathfrak{B}_c . For a system of n agents and an error threshold of $\varepsilon > 0$, the convergence time is bounded by $T(\varepsilon; n) = O(n^4 \log n / \varepsilon)$.*

4.2 Temporal complexity of the MaxMetropolis algorithm

To prove Theorem 3, we need to introduce a few technical results borrowed from [5], where they are given a more general and detailed exposition. In the following, we denote by $\sigma(-)$ the sample standard deviation: $\sigma(\mathbf{x}) := \sqrt{\sum_i (x_i - \langle \mathbf{x} \rangle)^2}$. The crux of the proof is to dominate $\sigma(\mathbf{x}(t))$ with a geometrically decreasing sequence, taking care when handling matrices with possibly negative entries.

► **Lemma 4.** *For any vector $\mathbf{v} \in \mathbb{R}^n$, we have*

$$\sqrt{2/n} \sigma(\mathbf{v}) \leq \text{diam } \mathbf{v} \leq 2 \sigma(\mathbf{v}). \quad (8)$$

The inequalities are strict if, and only if, the vectors \mathbf{v} and $\mathbf{1}$ are independent.

Proof. Developing the definition of the standard deviation, we have $\sigma(\mathbf{v}) = \sqrt{\frac{1}{2} \sum_{i \neq j} (v_i - v_j)^2}$, which yields the left-hand side inequality. Moreover, without loss of generality we can assume $\langle \mathbf{v} \rangle = 0$, in which case $\sigma(\mathbf{v}) = \|\mathbf{v}\|$; the right-hand side inequality then follows from the classic bounds $\text{diam } - \leq 2 \|\cdot\|_\infty$ and $\|\cdot\|_\infty \leq \|\cdot\|$. ◀

The following lemma is a restatement of a standard variational characterization of the eigenvalues of the matrix $\mathbf{I} - \mathbf{A}^\top \mathbf{A}$; see e.g., [11] for an in-depth treatment of the question.

► **Lemma 5.** *Let \mathbf{A} denote a doubly stochastic matrix, irreducible and with positive diagonal entries. For any vector \mathbf{v} , we have*

$$\sigma(\mathbf{A}\mathbf{v}) \leq \sqrt{1 - \gamma_{\mathbf{A}^\top \mathbf{A}}} \sigma(\mathbf{v}); \quad (9)$$

in the particular case where \mathbf{A} is symmetric, we have $\sigma(\mathbf{A}\mathbf{v}) \leq (1 - \gamma_{\mathbf{A}}) \sigma(\mathbf{v})$.

Finally, we will rely on the following spectral bound, given in [25, Lemma 9].

► **Lemma 6.** *Let \mathbf{A} be a stochastic matrix, with smallest positive entry α . If \mathbf{A} is symmetric, irreducible, and has positive diagonal entries, then we have*

$$\gamma_{\mathbf{A}} \geq \frac{\alpha}{n(n-1)}. \quad (10)$$

With Lemmas 4–6, we can turn to the proof of Theorem 3.

Proof of Theorem 3. Let us fix a dynamic graph $\mathbb{G} \in \mathfrak{B}_C$ with $n \geq 2$ vertices, and define

$$\begin{aligned} \overline{\deg}_i(t) &:= \max_{\tau \leq t} \deg_i \tau, & \overline{\deg}_i &:= \sup_{t \geq 1} \overline{\deg}_i(t), & \overline{\deg}_{\mathbb{G}} &:= \max_{i \in [n]} \overline{\deg}_i, & \text{and} \\ \mathcal{K} &:= \{t \geq 1 \mid \exists i: \overline{\deg}_i(t-1) < \overline{\deg}_i(t)\}, \end{aligned} \quad (11)$$

where by convention we set $\deg_i(0) = 2$ so that the set \mathcal{K} is properly defined. By definition, each sequence $\overline{\deg}_i(t)$ is weakly increasing with t , and has $\overline{\deg}_i$ for limit. Since $\deg_i(t) \leq n$, there are at most $\overline{\deg}_i$ rounds with $\overline{\deg}_i(t-1) < \overline{\deg}_i(t)$. The set \mathcal{K} is therefore finite, with cardinal $\delta := |\mathcal{K}| \leq \sum_i \overline{\deg}_i$. We let $t^* := \max \mathcal{K} + 1$; by construction, in all rounds $t \geq t^*$ we have $\overline{\deg}_i(t) = \overline{\deg}_i$.

By an immediate induction, we see that, in any execution of the MaxMetropolis algorithm over the dynamic communication graph \mathbb{G} , the sequence of estimate vectors satisfies the recurrence relation $\mathbf{x}(t) = \mathbf{A}(t) \mathbf{x}(t-1)$, where the affine MaxMetropolis matrix $\mathbf{A}(t)$ is given for off-diagonal entries $i \neq j$ by

$$A_{ij}(t) = \begin{cases} \frac{1}{\max(\overline{\deg}_i(t-1), \overline{\deg}_j(t-1))} & j \in \mathcal{N}_i(t) \\ 0 & j \notin \mathcal{N}_i(t), \end{cases} \quad (12)$$

and $\mathbf{x}(0) = (\mu_1, \dots, \mu_n)$ is given by the input values of the execution.

Equation (12), shows that the affine matrix $\mathbf{A}(t)$ is symmetric, and thus for any vector \mathbf{v} we have $\langle \mathbf{A}(t)\mathbf{v} \rangle = \langle \mathbf{v} \rangle$. This is true for all $t \geq 1$, and so $\langle \mathbf{x}(t) \rangle = \bar{\mu}$ is an invariant of the execution. If we show asymptotic consensus, then the consensus value is necessarily the initial average $\bar{\mu}$.

As a result of the Metropolis-Hastings symmetrization, the diagonal entries of the matrix $\mathbf{A}(t)$ satisfy

$$A_{ii}(t) \geq 1 - \frac{\deg_i(t) - 1}{\deg_i(t-1)}, \quad (13)$$

which gives in particular $A_{ii}(t) \geq 1/n$ when $t \notin \mathcal{K}$. The vector sequence $(\mathbf{x}(t))_{t \geq t^*}$ thus satisfies the assumptions of Proposition 2 for the uniform convexity parameter $\alpha = 1/n$, and so $\mathbf{x}(t)$ converges to a consensus vector. As already discussed, the limit value is necessarily

10:12 Computing Outside the Box

the initial average $\bar{\mu}$, and the system achieves asymptotic average consensus. This holds for any dynamic graph $\mathbb{G} \in \mathfrak{B}_c$ and arbitrary input values μ_1, \dots, μ_n , and thus MaxMetropolis is an average consensus algorithm for the class \mathfrak{B}_c .

It remains to show the polynomial convergence bound $T(\varepsilon; n) = O(n^4 \log n/\varepsilon)$. We start with the remark that the diagonal entry $A_{ii}(t)$ can be negative in a round t during which $\overline{\deg}_i(t) > \overline{\deg}_i(t-1)$. Because of this, the estimate $x_i(t)$ might end up outside the range of the previous estimates $\{x_1(t), \dots, x_n(t)\}$. As a consequence, rounds $t \in \mathcal{K}$ are “bad” rounds, where the system may move away from consensus, delaying the eventual convergence. In the class \mathfrak{B}_c , there is no uniform upper bound on the value of t^* , and such convexity-breaking rounds may occur arbitrarily late in the execution. Our challenge is therefore to show that, in finite time, the system reaches a given degree of agreement which cannot be undone in later “bad” rounds. We do this by accounting, from the start, the total delay that can be accrued in rounds $t \in \mathcal{K}$.

We follow the variations of the sample standard deviation $S(t) := \sigma(\mathbf{x}(t))$ from one round to the next, distinguishing on whether $t \in \mathcal{K}$ or not.

Case $t \notin \mathcal{K}$. By Equation (13), the irreducible matrix $\mathbf{A}(t)$ has positive diagonal entries, and thus has a positive spectral gap. By Lemma 5, we have

$$\forall t \notin \mathcal{K}: S(t) \leq (1 - \gamma_{\mathbf{A}(t)}) \cdot S(t-1). \quad (14)$$

Case $t \in \mathcal{K}$. Here, the matrix $\mathbf{A}(t)$ may have negative diagonal entries. It need not be a stochastic matrix, and indeed its spectral radius $\rho_{\mathbf{A}(t)}$ is possibly greater than 1. However, as a symmetric matrix, the matrix $\mathbf{A}(t)$ is diagonalizable, and thus we have $\|\mathbf{A}(t)\mathbf{v}\| \leq \rho_{\mathbf{A}(t)} \cdot \|\mathbf{v}\|$ for any vector \mathbf{v} . For the particular case $\mathbf{v} = \mathbf{x}(t-1) - \bar{\mu}\mathbf{1}$, this results in

$$\forall t \in \mathcal{K}: S(t) \leq \rho_{\mathbf{A}(t)} \cdot S(t-1). \quad (15)$$

Equation (15) actually holds for all $t \geq 1$, but it is strictly worse than Equation (14) for rounds $t \notin \mathcal{K}$.

Thus we let

$$\kappa(t) := \begin{cases} \rho_{\mathbf{A}(t)} & t \in \mathcal{K}, \\ 1 - \gamma_{\mathbf{A}(t)} & t \notin \mathcal{K}, \end{cases} \quad (16)$$

and we can summarize Equations (14) and (15) by $\forall t \geq 1: S(t) \leq \kappa(t) \cdot S(t-1)$. By induction, we then have $S(t) \leq \prod_{\tau \leq t} \kappa(\tau) \cdot S(0)$, and, applying Lemma 4 twice, we get

$$\forall t \geq 1: \text{diam } \mathbf{x}(t) \leq 2\sqrt{n} \prod_{\tau \leq t} \kappa(\tau) \cdot \text{diam } \boldsymbol{\mu}. \quad (17)$$

We are interested in the asymptotic behavior of $2\sqrt{n} \prod_{\tau \leq t} \kappa(\tau)$.

In order to bound the spectral radius $\rho_{\mathbf{A}(t)}$, we let $\nu_{t,i} := 1 - \min(0, A_{ii}(t))$, and $\nu_t := \max_i \nu_{t,i}$. Let us show that $\rho_{\mathbf{A}(t)} \leq \nu_t^2$: pick any eigenvalue $\lambda \in \text{Sp } \mathbf{A}(t)$. By construction, the quantity $\left(1 + \frac{\lambda-1}{\nu_t}\right)$ is an eigenvalue of the stochastic matrix $\frac{1}{\nu_t}(\mathbf{A}(t) + (\nu_t - 1)\mathbf{I})$, and so is less than 1 in absolute value. We have $1 - 2\nu_t \leq \lambda \leq 1$, and so $|\lambda| \leq 2\nu_t - 1$. Using the basic inequality $x^2 - 2x + 1 \geq 0$, we have $|\lambda| \leq \nu_t^2$, and therefore $\rho_{\mathbf{A}(t)} \leq \nu_t^2$ since this holds for any $\lambda \in \text{Sp } \mathbf{A}(t)$.

For any $t \in \mathcal{K}$ and $i \in [n]$, we have $\sum_{j \neq i} A_{ij}(t) \leq \frac{\deg_i(t)-1}{\deg_i(t-1)} \leq \frac{\overline{\deg}_i(t)}{\overline{\deg}_i(t-1)}$. Since $\overline{\deg}_i(t)$ is weakly increasing with t , we have in turn $\nu_{t,i} \leq \frac{\overline{\deg}_i(t)}{\overline{\deg}_i(t-1)}$, from here we have

$$\begin{aligned} \prod_{t \in \mathcal{K}} \rho_{\mathbf{A}(t)} &\leq \left(\prod_{t \in \mathcal{K}} \max_{i \in [n]} \nu_{t,i} \right)^2 && \left. \begin{array}{l} \nu_{t,i} \geq 1 \\ \nu_{t,i} \leq \frac{\overline{\deg}_i(t)}{\overline{\deg}_i(t-1)} \end{array} \right\} \\ &\leq \left(\prod_{t \in \mathcal{K}} \prod_{i \in [n]} \nu_{t,i} \right)^2 && \\ &\leq \left(\prod_{i \in [n]} \prod_{t \in \mathcal{K}} \frac{\overline{\deg}_i(t)}{\overline{\deg}_i(t-1)} \right)^2 && \left. \begin{array}{l} \nu_{t,i} \leq \frac{\overline{\deg}_i(t)}{\overline{\deg}_i(t-1)} \\ \overline{\deg}_i(t) = \overline{\deg}_i(t-1) \text{ when } t \notin \mathcal{K} \end{array} \right\} \\ &\leq \left(\prod_{i \in [n]} \prod_{t \geq 1} \frac{\overline{\deg}_i(t)}{\overline{\deg}_i(t-1)} \right)^2 && \\ &= \left(\prod_{i \in [n]} \frac{\overline{\deg}_i}{2} \right)^2 = 2^{-2n} \varpi^2, \end{aligned}$$

where $\varpi := \prod_{i \in [n]} \overline{\deg}_i$.

From here, we let $\gamma := \inf_{t \notin \mathcal{K}} \gamma_{\mathbf{A}(t)}$, and we have

$$\begin{aligned} \prod_{\tau \leq t} \kappa(\tau) &= \left(\prod_{\tau \in [1,t] \cap \mathcal{K}} \kappa(\tau) \right) \left(\prod_{\tau \in [1,t] \setminus \mathcal{K}} \kappa(\tau) \right) && \left. \begin{array}{l} \kappa(\tau \in \mathcal{K}) \geq 1 \\ \kappa(\tau \notin \mathcal{K}) \leq 1 - \gamma_{\mathbf{A}(t)} \end{array} \right\} \\ &\leq \left(\prod_{\tau \in \mathcal{K}} \rho_{\mathbf{A}(t)} \right) \left(\prod_{\tau \in [1,t] \setminus \mathcal{K}} (1 - \gamma_{\mathbf{A}(t)}) \right) && \\ &\leq 2^{-2n} \varpi^2 (1 - \gamma)^{t-\delta}. \end{aligned}$$

As a consequence, given any error threshold $\varepsilon > 0$, the estimates are contained in a ball of diameter $(\varepsilon \cdot \text{diam } \boldsymbol{\mu})$ at the latest in round $t_\varepsilon \leq \delta + \gamma^{-1} \log(2^{-2n+1} \varpi^2 \sqrt{n}/\varepsilon)$. From Lemma 6, we have $\gamma^{-1} \leq n(n-1) \overline{\deg}_{\mathbb{G}}$, and using $\varpi := \prod_{i \in [n]} \overline{\deg}_i$ and $\delta := |\mathcal{K}| \leq \sum_{i \in [n]} \overline{\deg}_i - 2n$, we get:

$$t_\varepsilon \leq \sum_i \overline{\deg}_i - 2n + n(n-1) \overline{\deg}_{\mathbb{G}} \left(2 \sum_i \log \overline{\deg}_i - (2n-1) \log 2 - \log \varepsilon \right), \quad (18)$$

which, using the fact that $\overline{\deg}_i \leq n$, finally gives us $t_\varepsilon = O(n^4 \log n/\varepsilon)$. \blacktriangleleft

Compared to the $O(n^2 \log n/\varepsilon)$ convergence time of the Metropolis rule, the latter asymptotic bound is worse by a factor $n \overline{\deg}_{\mathbb{G}}$. From the proof, we can give a rough analysis of this factors: the factor n represents the delay due to broken convexity, as each agent individually induces a delay of $\log \overline{\deg}_i$. The factor $\overline{\deg}_{\mathbb{G}}$ comes from the fact that, whereas the Metropolis rule always selects the *best* possible off-diagonal weights – that is, the largest ones, – the MaxMetropolis rule makes conservative choices so as to allow for a decentralized algorithmic implementation that only breaks convexity finitely many times.

Improvements to the MaxMetropolis approach, based for example on adjusting the parameters q_i downwards in pursuit of faster mixing, must therefore be considered with extreme care, as gains due to larger weights might result in greater delays due to broken convergence.

5 Conclusion

In this paper, we have presented the MaxMetropolis algorithm, a parsimonious distributed algorithm for average consensus that operates in polynomial time over connected bidirectional dynamic networks, without resorting to any centralized crutch like unique identifiers, a designated leader, or global information on the network.

Our solution has many potential uses, given that average consensus primitives underpin many applications studied in distributed control. In contrast with the classic approaches used in this domain, we take an algorithmic stance, grounded in the theory of anonymous computation [1, 2, 17] and of the algorithmic study of dynamic networks [20]. We argue that the fundamental convex recurrence rule for average consensus, namely, the Metropolis rule, cannot be implemented in a fully distributed and decentralized setting when the network is subject to unpredictable change. Our solution consists in relaxing the *convexity* constraint, resulting in an *affine* recurrence rule for average consensus that is algorithmically implementable in any networked multi-agent system with a time-varying communication graph, under the sole constraint of bidirectional links and permanent connectivity.

In the long version of our paper, we will relax the latter assumption and show that $(B \geq 1)$ -bounded connectivity – where it is only each *matrix product* $\mathbf{A}(t + B - 1) \cdots \mathbf{A}(t)$ that is assumed irreducible – only delays our convergence bound by a factor B . An open question is whether one can design a fully decentralized average consensus algorithm that doesn't break the convex hull of the estimates, or whether that is impossible.

References

- 1 Dana Angluin. Local and global properties in networks of processors (extended abstract). In R. E. Miller, S Ginsburg, W. A. Burkhard, and R. J. Lipton, editors, *Proceedings of the twelfth annual ACM symposium on Theory of computing - STOC '80*, pages 82–93. ACM Press, 1980. doi:10.1145/800141.804655.
- 2 Paolo Boldi and Sebastiano Vigna. An effective characterization of computability in anonymous networks. In Jennifer Welch, editor, *DISC 2001: Distributed Computing*, volume 2180 of *Lecture Notes in Computer Science*, pages 33–47. Springer Berlin Heidelberg, 2001. doi:10.1007/3-540-45414-4_3.
- 3 Florence Bénézit, Vincent D. Blondel, Patrick Thiran, John N. Tsitsiklis, and Martin Vetterli. Weighted gossip: Distributed averaging using non-doubly stochastic matrices. In *2010 IEEE International Symposium on Information Theory*, pages 1753–1757, June 2010. doi:10.1109/ISIT.2010.5513273.
- 4 Themistoklis Charalambous, Michael G. Rabbat, Mikael Johansson, and Christoforos N. Hadjicostis. Distributed Finite-Time Computation of Digraph Parameters: Left-Eigenvector, Out-Degree and Spectrum. *IEEE Transactions on Control of Network Systems*, 3(2):137–148, 2016. doi:10.1109/TCNS.2015.2428411.
- 5 Bernadette Charron-Bost. Geometric Bounds for Convergence Rates of Averaging Algorithms. *Information and Computation*, 2022. (To appear). arXiv:2007.04837.
- 6 Bernadette Charron-Bost and Patrick Lambein-Monette. Randomization and quantization for average consensus. In *2018 IEEE Conference on Decision and Control (CDC)*, pages 3716–3721. IEEE, December 2018. doi:10.1109/CDC.2018.8619817.
- 7 Samprit Chatterjee and Eugene Seneta. Towards Consensus: Some Convergence Theorems on Repeated Averaging. *Journal of Applied Probability*, 14(1):89–97, 1977. doi:10.2307/3213262.
- 8 George Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2):279–301, 1989. doi:10.1016/0743-7315(89)90021-X.
- 9 Louis Penet de Monterno, Bernadette Charron-Bost, and Stephan Merz. Synchronization modulo k in dynamic networks. In *Stabilization, Safety, and Security of Distributed Systems - 23rd International Symposium, SSS 2021, Virtual Event, November 17-20, 2021, Proceedings*, volume 13046 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 2021. doi:10.1007/978-3-030-91081-5_28.
- 10 Morris H. DeGroot. Reaching a Consensus. *Journal of the American Statistical Association*, 69(345):118–121, 1974. doi:10.2307/2285509.

- 11 Persi Diaconis and Daniel Stroock. Geometric bounds for eigenvalues of markov chains. *The Annals of Applied Probability*, 1(1):36–61, February 1991. doi:10.1214/aoap/1177005980.
- 12 Michael Dinitz, Jeremy Fineman, Seth Gilbert, and Calvin Newport. Load balancing with bounded convergence in dynamic networks. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, Atlanta, GA, USA, 2017. IEEE. doi:10.1109/INFOCOM.2017.8057000.
- 13 Alejandro D. Dominguez-Garcia, Stanton T. Cady, and Christoforos N. Hadjicostis. Decentralized optimal dispatch of distributed energy resources. In *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*, pages 3688–3693. IEEE, December 2012. ZSCC: 0000166. doi:10/ggd8nx.
- 14 Balazs Gerencser and Julien M. Hendrickx. Push-sum with transmission failures. *IEEE Transactions on Automatic Control*, 64(3):1019–1033, March 2019. doi:10.1109/TAC.2018.2836861.
- 15 Wilfred Keith Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, April 1970. doi:10.1093/biomet/57.1.97.
- 16 Julien M. Hendrickx, Alex Olshevsky, and John N. Tsitsiklis. Distributed anonymous discrete function computation. *IEEE Transactions on Automatic Control*, 56(10):2276–2289, October 2011. doi:10.1109/TAC.2011.2163874.
- 17 Julien M. Hendrickx and John N. Tsitsiklis. Fundamental limitations for anonymous distributed systems with broadcast communications. In *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 9–16. IEEE, September 2015. doi:10.1109/ALLERTON.2015.7446980.
- 18 Ali Jadbabaie, Jie Lin, and A. Stephen Morse. Coordination of groups of mobile autonomous agents using nearest neighbor rules. *IEEE Transactions on Automatic Control*, 48(6):988–1001, 2003. doi:10.1109/TAC.2003.812781.
- 19 David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.*, pages 482–491, October 2003. doi:10/fcmmkg.
- 20 Fabian Kuhn, Nancy Lynch, and Rotem Oshman. Distributed computation in dynamic networks. In *Proceedings of the 42nd ACM symposium on Theory of computing - STOC '10*, page 513. ACM Press, 2010. doi:10.1145/1806689.1806760.
- 21 Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, March 1953. doi:10.2172/4390578.
- 22 Luc Moreau. Stability of multiagent systems with time-dependent communication links. *IEEE Transactions on Automatic Control*, 50(2):169–182, 2005. doi:10.1109/TAC.2004.841888.
- 23 Damon Mosk-Aoyama and Devavrat Shah. Computing separable functions via gossip. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing, PODC '06*, pages 113–122. ACM Press, July 2006. doi:10.1145/1146381.1146401.
- 24 Angelia Nedić and Alex Olshevsky. Distributed optimization over time-varying directed graphs. *IEEE Transactions on Automatic Control*, 60(3):601–615, 2014. doi:10/f63582.
- 25 Angelia Nedić, Alex Olshevsky, Asuman Ozdaglar, and John N. Tsitsiklis. On Distributed Averaging Algorithms and Quantization Effects. *IEEE Transactions on Automatic Control*, 54(11):2506–2517, 2009. doi:10.1109/TAC.2009.2031203.
- 26 Angelia Nedić, Alex Olshevsky, and Michael G. Rabbat. Network Topology and Communication-Computation Tradeoffs in Decentralized Optimization. *Proceedings of the IEEE*, 106(5):953–976, 2018. doi:10.1109/JPROC.2018.2817461.
- 27 Reza Olfati-Saber and Jeff S. Shamma. Consensus filters for sensor networks and distributed sensor fusion. In *Proceedings of the 44th IEEE Conference on Decision and Control*, pages 6698–6703, December 2005. doi:10/c338d4.

- 28 Alex Olshevsky. Linear Time Average Consensus and Distributed Optimization on Fixed Graphs. *SIAM Journal on Control and Optimization*, 55(6):3990–4014, 2017. doi:10.1137/16M1076629.
- 29 Alex Olshevsky and John N. Tsitsiklis. Convergence Speed in Distributed Consensus and Averaging. *SIAM Review*, 53(4):747–772, 2011. doi:10.1137/110837462.
- 30 W. Ren. Consensus strategies for cooperative control of vehicle formations. *IET Control Theory & Applications*, 1(2):505–512, 2007. doi:10.1049/iet-cta:20050401.
- 31 Shreyas Sundaram and Christoforos N. Hadjicostis. Finite-Time Distributed Consensus in Graphs with Time-Invariant Topologies. In *2007 American Control Conference*, pages 711–716, 2007. doi:10.1109/ACC.2007.4282726.
- 32 Ichiro Suzuki and Masafumi Yamashita. Distributed Anonymous Mobile Robots: Formation of Geometric Patterns. *SIAM Journal on Computing*, 28(4):1347–1363, 1999. doi:10.1137/S009753979628292X.
- 33 John N. Tsitsiklis. *Problems in Decentralized Decision Making and Computation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, 1984. URL: <https://www.mit.edu/~jnt/Papers/PhD-84-jnt.pdf>.
- 34 John N. Tsitsiklis, Dimitri P. Bertsekas, and Michael Athans. Distributed Asynchronous Deterministic and Stochastic Gradient Optimization Algorithms. *IEEE Transactions on Automatic Control*, 31(9):803–812, 1986. doi:10.1109/TAC.1986.1104412.
- 35 Lin Xiao and Stephen Boyd. Fast linear iterations for distributed averaging. *Systems & Control Letters*, 53(1):65–78, 2004. doi:10.1016/j.sysconle.2004.02.022.
- 36 Lin Xiao, Stephen Boyd, and Seung-Jean Kim. Distributed average consensus with least-mean-square deviation. *Journal of Parallel and Distributed Computing*, 67(1):33–46, 2007. doi:10/bmq2t4.
- 37 Lin Xiao, Stephen Boyd, and Sanjay Lall. A Scheme for Robust Distributed Sensor Fusion Based on Average Consensus. In *Fourth International Symposium on Information Processing in Sensor Networks, 2005.*, pages 63–70, Los Angeles, CA, USA, 2005. IEEE. doi:10.1109/IPSN.2005.1440896.

Fast and Succinct Population Protocols for Presburger Arithmetic

Philipp Czerner   

Department of Informatics, Technische Universität München, Germany

Roland Guttenberg  

Department of Informatics, Technische Universität München, Germany

Martin Helfrich   

Department of Informatics, Technische Universität München, Germany

Javier Esparza   

Department of Informatics, Technische Universität München, Germany

Abstract

In their 2006 seminal paper in Distributed Computing, Angluin et al. present a construction that, given any Presburger predicate as input, outputs a leaderless population protocol that decides the predicate. The protocol for a predicate of size m (when expressed as a Boolean combination of threshold and remainder predicates with coefficients in binary) runs in $\mathcal{O}(m \cdot n^2 \log n)$ expected number of interactions, which is almost optimal in n , the number of interacting agents. However, the number of states of the protocol is exponential in m . This is a problem for natural computing applications, where a state corresponds to a chemical species and it is difficult to implement protocols with many states. Blondin et al. described in STACS 2020 another construction that produces protocols with a polynomial number of states, but exponential expected number of interactions. We present a construction that produces protocols with $\mathcal{O}(m)$ states that run in expected $\mathcal{O}(m^7 \cdot n^2)$ interactions, optimal in n , for all inputs of size $\Omega(m)$. For this, we introduce population computers, a carefully crafted generalization of population protocols easier to program, and show that our computers for Presburger predicates can be translated into fast and succinct population protocols.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases population protocols, fast, succinct, population computers

Digital Object Identifier 10.4230/LIPIcs.SAND.2022.11

Related Version *Full Version*: <https://arxiv.org/abs/2202.11601v2> [11]

Funding This work was supported by an ERC Advanced Grant (787367: PaVeS) and by the Research Training Network of the Deutsche Forschungsgemeinschaft (DFG) (378803395: ConVeY).

Acknowledgements We thank the anonymous reviewers for many helpful remarks. In particular, one remark led to Lemma 11, which in turn led to a nicer formulation of Theorem 2, one of our main results.

1 Introduction

Population protocols [4, 5] are a model of computation in which indistinguishable, mobile finite-state agents, randomly interact in pairs to decide whether their initial configuration satisfies a given property, modelled as a predicate on the set of all configurations. The decision is taken by *stable consensus*; eventually all agents agree on whether the property holds or not, and never change their mind again. Population protocols are very close to chemical reaction networks, a model in which agents are molecules and interactions are chemical reactions.



© Philipp Czerner, Roland Guttenberg, Martin Helfrich, and Javier Esparza;
licensed under Creative Commons License CC-BY 4.0

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).

Editors: James Aspnes and Othon Michail; Article No. 11; pp. 11:1–11:17

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In a seminal paper, Angluin et al. proved that population protocols decide exactly the predicates definable in Presburger arithmetic (PA) [7]. One direction of the result is proved in [5] by means of a construction that takes as input a Presburger predicate and outputs a protocol that decides it. The construction uses the quantifier elimination procedure for PA: every Presburger formula φ can be transformed into an equivalent boolean combination of *threshold* predicates of the form $\vec{a} \cdot \vec{x} \geq c$ and *remainder* predicates of the form $\vec{a} \cdot \vec{x} \equiv_m c$, where \vec{a} is an integer vector, c and m are integers, and \equiv_m denotes congruence modulo m [14]. Slightly abusing language, we call the set of these boolean combinations *quantifier-free Presburger arithmetic* (QFPA).¹ Using that PA and QFPA have the same expressive power, Angluin et al. first construct protocols for all threshold and remainder predicates, and then show that the predicates computed by protocols are closed under negation and conjunction.

The two fundamental parameters of a protocol are the expected number of interactions until a stable consensus is reached, and the number of states of each agent. The expected number of interactions divided by the number of agents, also called the parallel execution time, is an adequate measure of the runtime of a protocol when interactions occur in parallel according to a Poisson process [6]. The number of states measures the complexity of an agent. In natural computing applications, where a state corresponds to a chemical species, it is difficult to implement protocols with many states.

Given a formula φ of QFPA, let m be the number of bits of the largest coefficient of φ in absolute value, and let s be the number of atomic formulas of φ , respectively. Let n be the number of agents participating in the protocol. The construction of [5] yields a protocol with $\mathcal{O}(s \cdot n^2 \log n)$ expected interactions. Observe that the protocol does not have a leader (an auxiliary agent helping the other agents to coordinate), and agents have a fixed number of states, independent of the size of the population. Under these assumptions, which are also the assumptions of this paper, every protocol for the majority predicate needs $\Omega(n^2)$ expected interactions [1], and so the construction is nearly optimal.² However, the number of states is $\Omega(2^{m+s})$, or $\Omega(2^{|\varphi|})$ in terms of the number $|\varphi|$ of bits needed to write φ with coefficients in binary. This is well beyond the only known lower bound, showing that for every construction there exist an infinite subset of predicates φ for which the construction produces protocols with $\Omega(|\varphi|^{1/4})$ states [9]. So the constructions of [5], and also those of [6, 3, 13], produce *fast* but *very large* protocols.

In [9, 8] Blondin et al. exhibit a construction that produces *succinct* protocols with $\mathcal{O}(\text{poly}(|\varphi|))$ states. However, they do not analyse their stabilisation time. We demonstrate that they run in $\Omega(2^n)$ expected interactions. Loosely speaking, the reason is the use of transitions that “revert” the effect of other transitions. This allows the protocol to “try out” different distributions of agents, retracing its steps until it hits the right one, but also makes it very slow. So [9, 8] produce *succinct* but *very slow* protocols.

Is it possible to produce protocols that are both *fast* and *succinct*? We give an affirmative answer. We present a construction that yields for every formula φ of QFPA a protocol with $\mathcal{O}(\text{poly}(|\varphi|))$ states and $\mathcal{O}(\text{poly}(|\varphi|) \cdot n^2)$ expected interactions. So our construction achieves optimal stabilisation time in n , and, at the same time, yields more succinct protocols than the construction of [8]. Moreover, for inputs of size $\Omega(|\varphi|)$ (a very mild constraint when agents are molecules), we obtain protocols with $\mathcal{O}(|\varphi|)$ states.

¹ Remainder predicates cannot be directly expressed in Presburger arithmetic without quantifiers.

² If the model is extended by allowing a *leader* (and one considers the slightly weaker notion of convergence time), or the number of states of an agent is allowed to grow with the population size, $\mathcal{O}(n \cdot \text{polylog}(n))$ interactions can be achieved [6, 3, 2, 13, 12].

Our construction relies on *population computers*, a carefully crafted generalization of the population protocol model of [5]. Population computers extend population protocols in three ways. First, they can exhibit certain *k-way interactions* between more than two agents. Second, they have a more flexible *output condition*, defined by an arbitrary function that assigns an output to every subset of states, instead of to every state.³ Finally, population computers can use *helpers*: auxiliary agents that, like leaders, help regular agents to coordinate themselves but whose number, contrary to leaders, is not known *a priori*. We exhibit succinct population computers for all Presburger predicates in which every run is finite, and show how to translate such population computers into fast and succinct population protocols.

Organization of the paper. We give preliminary definitions in Section 2 and introduce population computers in Section 3. Section 4 gives an overview of the rest of the paper and summarises our main results. Section 5 describes why previous constructions were either not succinct or slow. Section 6 describes population computers for every Presburger predicate. Section 7 converts these computers into succinct population protocols. Section 8 shows that the resulting protocols are also fast.

An extended version of this paper, containing the details of the constructions and all proofs, can be found at [11]. It contains several appendices. Appendix A completes the proofs of Section 5. For the other appendices, there is no one-to-one correspondence to sections of the main paper, instead they are grouped by the construction they analyse. Appendix B concerns the construction of Section 6, but also analyses speed. The four parts of our conversion process are analysed separately, in Appendices C, D, E and F. Appendix G combines the previous to prove the complete conversion theorem. Appendix H summarises the definitions for our speed analyses, and Appendix I contains minor technical lemmata.

2 Preliminaries

Multisets. Let E be a finite set. A multiset over E is a mapping $E \rightarrow \mathbb{N}$, and \mathbb{N}^E denotes the set of all multisets over E . We sometimes write multisets using set-like notation, e.g. $\{a, 2 \cdot b\}$ denotes the multiset v such that $v(a) = 1$, $v(b) = 2$ and $v(e) = 0$ for every $e \in E \setminus \{a, b\}$. The empty multiset $\{\}$ is also denoted \emptyset .

For $E' \subseteq E$, $v(E') := \sum_{e \in E'} v(e)$ is the number of elements in v that are in E' . The *size* of $v \in \mathbb{N}^E$ is $|v| := v(E)$. The *support* of $v \in \mathbb{N}^E$ is the set $\text{supp}(v) := \{e \in E \mid v(e) > 0\}$. If $E \subseteq \mathbb{Z}$, then we let $\text{sum}(v) := \sum_{e \in E} e \cdot v(e)$ denote the sum of all the elements of v . Given $u, v \in \mathbb{N}^E$, $u + v$ and $u - v$ denote the multisets given by $(u + v)(e) := u(e) + v(e)$ and $(u - v)(e) := u(e) - v(e)$ for every $e \in E$. The latter is only defined if $u \geq v$.

Multiset rewriting transitions. A *multiset rewriting transition*, or just a *transition*, is a pair $(r, s) \in \mathbb{N}^E \times \mathbb{N}^E$, also written $r \mapsto s$. A transition $t = (r, s)$ is *enabled* at $v \in \mathbb{N}^E$ if $v \geq r$, and its *occurrence* leads to $v' := v - r + s$, denoted $v \rightarrow_t v'$. We call $v \rightarrow_t v'$ a *step*. The multiset v is *terminal* if it does not enable any transition. An *execution* is a finite or infinite sequence v_0, v_1, \dots of multisets such that $v \rightarrow_{t_1} v_1 \rightarrow_{t_2} \dots$ for some sequence t_1, t_2, \dots of transitions. A multiset v' is *reachable* from v if there is an execution v_0, v_1, \dots, v_k with $v_0 = v$ and $v_k = v'$; we also say that the execution *leads from v to v'* . An execution is a *run* if it is infinite or it is finite and its last multiset is terminal. A run v_0, v_1, \dots is *fair* if it is finite, or it is infinite and for every multiset v , if v is reachable from v_i for infinitely many $i \geq 0$, then $v = v_j$ for some $j \geq 0$.

³ Other output conventions for population protocols have been considered [10].

Presburger arithmetic. Angluin et al. proved that population protocols decide exactly the predicates $\mathbb{N}^k \rightarrow \{0, 1\}$ definable in Presburger arithmetic, the first-order theory of addition, which coincide with the *semilinear* predicates [14]. Using the quantifier elimination procedure of Presburger arithmetic, every Presburger predicate can be represented as a Boolean combination of *threshold* and *remainder* predicates. A predicate $\varphi : \mathbb{N}^v \rightarrow \{0, 1\}$ is a threshold predicate if $\varphi(x_1, \dots, x_v) = (\sum_{i=1}^v a_i x_i \geq c)$, where $a_1, \dots, a_v, c \in \mathbb{Z}$, and a remainder predicate if $\varphi(x_1, \dots, x_v) = (\sum_{i=1}^v a_i x_i \equiv_m c)$, where $a_1, \dots, a_v \in \mathbb{Z}$, $m \geq 1$, $c \in \{0, \dots, m-1\}$, and $a \equiv_m b$ denotes that a is congruent to b modulo m . We call the set of these formulas *quantifier-free Presburger arithmetic*, or QFPA. The *size* of a predicate is the minimal number of bits of a formula of QFPA representing it, with coefficients written in binary.

3 Population Computers

Population computers are a generalization of population protocols that allows us to give very concise descriptions of our protocols for Presburger predicates.

Syntax. A *population computer* is a tuple $\mathcal{P} = (Q, \delta, I, O, H)$, where:

- Q is a finite set of *states*. Multisets over Q are called *configurations*.
- $\delta \subseteq \mathbb{N}^Q \times \mathbb{N}^Q$ is a set of multiset rewriting transitions $r \mapsto s$ over Q such that $|r| = |s| \geq 2$ and $|\text{supp}(r)| \leq 2$. Further, we require that δ is a partial function, so $s_1 = s_2$ for all r, s_1, s_2 with $(r, s_1), (r, s_2) \in \delta$. A transition $r \mapsto s$ is *binary* if $|r| = 2$. We call a population computer *binary* if every transition binary.
- $I \subseteq Q$ is a set of *input states*. An *input* is a configuration C such that $\text{supp}(C) \subseteq \text{supp}(I)$.
- $O : 2^Q \rightarrow \{0, 1, \perp\}$ is an *output function*. The *output* of a configuration C is $O(\text{supp}(C))$. An output function O is a *consensus output* if there is a partition $Q = Q_0 \cup Q_1$ of Q such that $O(Q') = 0$ iff $Q' \subseteq Q_0$, $O(Q') = 1$ iff $Q' \subseteq Q_1$, and $O(Q') = \perp$ otherwise.
- $H \in \mathbb{N}^{Q \setminus I}$ is a multiset of *helper agents* or just *helpers*. A *helper configuration* is a configuration C such that $\text{supp}(C) \subseteq \text{supp}(H)$ and $C \geq H$.

Graphical notation. We visualise population computers as Petri nets (see e.g. Figure 3). Places (circles) and transitions (squares) represent respectively states and transitions. To visualise configurations, we draw agents as tokens (smaller filled circles).

Semantics. Intuitively, a population computer decides which output (0 or 1) corresponds to an input C_I as follows. It adds to the agents of C_I an arbitrary helper configuration C_H of agents to produce the initial configuration $C_I + C_H$. Then it starts the computation and lets it stabilise to configurations of output 1 or output 0. Formally, the *initial configurations* of \mathcal{P} for input C_I are all configurations of the form $C_I + C_H$ for some helper configuration C_H . A run $C_0 C_1 \dots$ *stabilises to* b if there exists an $i \geq 0$ such that $O(\text{supp}(C_i)) = b$ and C_i only reaches configurations C' with $O(\text{supp}(C')) = b$. An input C_I *has output* b if for every initial configuration $C_0 = C_I + C_H$, every fair run starting at C_0 stabilises to b . A population computer \mathcal{P} *decides* a predicate $\varphi : \mathbb{N}^I \rightarrow \{0, 1\}$ if every input C_I has output $\varphi(C_I)$.

Terminating and bounded computers. A population computer is *bounded* if no run starting at an initial configuration C is infinite, and *terminating* if no fair run starting at C is infinite. Observe that bounded population computers are terminating.

Size and adjusted size. Let $\mathcal{P} = (Q, \delta, I, O, H)$ be a population computer. We assume that O is described as a boolean circuit with $\text{size}(O)$ gates. For every transition $t = (r \mapsto s)$ let $|t| := |r|$. The *size* of \mathcal{P} is $\text{size}(\mathcal{P}) := |Q| + |H| + \text{size}(O) + \sum_{t \in \delta} |t|$. If \mathcal{P} is binary, then (as for population protocols) we do not count the transitions and define the *adjusted size* $\text{size}_2(\mathcal{P}) := |Q| + |H| + \text{size}(O)$. Observe that both the size of a transition and the size of the helper multiset are the number of elements, i.e. the size in unary, strengthening our later result about the existence of succinct population computers.

Population protocols. A population computer $\mathcal{P} = (Q, \delta, I, O, H)$ is a *population protocol* if it is binary, has no helpers ($H = \emptyset$), and O is a consensus output. It is easy to see that this definition coincides with the one of [5]. The speed of a binary population computer with no helpers, and so in particular of a population protocol, is defined as follows. We assume a probabilistic execution model in which at configuration C two agents are picked uniformly at random and execute a transition, if possible, moving to a configuration C' (by assumption they enable at most one transition). This is called an *interaction*. Repeating this process, we generate a *random execution* $C_0 C_1 \dots$. We say that the execution *stabilises* at time t if C_t reaches only configurations C' with $O(\text{supp}(C')) = O(\text{supp}(C_t))$, and we say that \mathcal{P} *decides* φ *within* T *interactions* if it decides φ and $\mathbb{E}(t) \leq T$. See e.g. [6] for more details.

Population computers vs. population protocols. Population computers generalise population protocols in three ways:

- They have non-binary transitions, but only those in which the interacting agents populate at most two states. For example, $\langle p, p, q \rangle \mapsto \langle p, q, o \rangle$ (which in the following is written simply as $p, p, q \mapsto p, q, o$) is allowed, but $p, q, o \mapsto p, p, q$ is not.
- They use a multiset H of auxiliary helper agents, but the addition of more helpers does not change the output of the computation. Intuitively, contrary to the case of leaders, agents do not know any upper bound on the number of helpers, and so the protocol cannot rely on this bound for correctness or speed.
- They have a more flexible output condition. Loosely speaking, population computers accept by stabilising the population to an accepting set of states, instead of to a set of accepting states.

4 Overview and Main Results

Given a predicate $\varphi \in QFPA$ over variables x_1, \dots, x_v , the rest of this paper shows how to construct a fast and succinct population protocol deciding φ . First, Section 5 gives an overview of previous constructions and explains why they are not fast or not succinct. Then we proceed in five steps:

1. Construct the predicate $\text{double}(\varphi) \in QFPA$ over variables $x_1, \dots, x_v, x'_1, \dots, x'_v$ by syntactically replacing every occurrence of x_i in φ by $x_i + 2x'_i$. For example, if $\varphi = (x - y \geq 0)$ then $\text{double}(\varphi) = (x + 2x' - y - 2y' \geq 0)$. Observe that $|\text{double}(\varphi)| \in \mathcal{O}(|\varphi|)$.
2. Construct a succinct bounded population computer \mathcal{P} deciding $\text{double}(\varphi)$.
3. Convert \mathcal{P} into a succinct population protocol \mathcal{P}' deciding φ for inputs of size $\Omega(|\varphi|)$.
4. Prove that \mathcal{P}' runs within $\mathcal{O}(n^3)$ interactions.
5. Use a refined running-time analysis to prove that \mathcal{P}' runs within $\mathcal{O}(n^2)$ interactions.

Section 6 constructs bounded population computers for all predicates $\varphi \in QFPA$. This allows us to conduct steps 1 and 2. More precisely, the section proves:

► **Theorem 1.** *For every predicate $\varphi \in QFPA$ there exists a bounded population computer of size $\mathcal{O}(|\varphi|)$ that decides φ .*

Section 7 proves the following conversion theorem (steps 3 and 4).

► **Theorem 2.** *Every bounded population computer of size m deciding $\text{double}(\varphi)$ can be converted into a terminating population protocol with $\mathcal{O}(m^2)$ states which decides φ in at most $\mathcal{O}(f(m)n^3)$ interactions for inputs of size $\Omega(m)$, for some function f .*

Section 8 introduces α -rapid population computers, where $\alpha \geq 1$ is a certain parameter, and uses a more detailed analysis to show that the population protocols of Theorem 2 are in fact smaller and faster (step 5):

► **Theorem 3.**

- (a) *The population computers constructed in Theorem 1 are $\mathcal{O}(|\varphi|^3)$ -rapid.*
- (b) *Every α -rapid population computer of size m deciding $\text{double}(\varphi)$ can be converted into a terminating population protocol with $\mathcal{O}(m)$ states that decides φ in $\mathcal{O}(\alpha m^4 n^2)$ interactions for inputs of size $\Omega(m)$.*

The restriction to inputs of size $\Omega(m)$ is very mild. Moreover, it can be lifted using a technique of [8], at the price of adding additional states (and at no cost regarding asymptotic speed, since the speed of the new protocol only changes for inputs of size $\mathcal{O}(m)$):

► **Corollary 4.** *For every $\varphi \in QFPA$ there exists a terminating population protocol with $\mathcal{O}(\text{poly}(|\varphi|))$ states that decides φ in $\mathcal{O}(f(|\varphi|)n^2)$ interactions, for a function f .*

It is known that the majority predicate can only be decided in $\Omega(n^2)$ interactions by population protocols [1], so — as a general construction — our result is optimal w.r.t. time. Regarding space, an $\Omega(|\varphi|^{1/4})$ lower bound was shown in [9], leaving a polynomial gap.

5 Previous Constructions: Angluin et al. and Blondin et al.

The population protocols for a quantifier free Presburger predicate φ constructed in [5] are not *succinct*, i.e. do not have $\mathcal{O}(|\varphi|^a)$ states for any constant a , and those of [8] are not *fast*, i.e. do not have speed $\mathcal{O}(|\varphi|^a n^b)$ for any constants a, b . We explain why with the help of some examples.

► **Example 5.** Consider the protocol of [5] for the predicate $\varphi = (x - y \geq 2^d)$. The states are the triples (ℓ, b, u) where $\ell \in \{A, P\}$, $b \in \{Y, N\}$ and $-2^d \leq u \leq 2^d$. Intuitively, ℓ indicates whether the agent is active (A) or passive (P), b indicates whether it currently believes that φ holds (Y) or not (N), and u is the agent's wealth, which can be negative. Agents for input x are initially in state $(A, N, 1)$, and agents for y in $(A, N, -1)$. If two passive agents meet their encounter has no effect. If at least one agent is active, then the result of the encounter is given by the transition $(*, *, u), (*, *, u') \mapsto (A, b, q), (P, b, r)$ where $b = Y$ if $u + u' \geq 2^d$ else N ; $q = \max(-2^d, \min(2^d, u + u'))$; and $r = (u + u') - q$. The protocol stabilises after $\mathcal{O}(n^2 \log n)$ expected interactions [5], but it has $2^{d+1} + 1$ states, exponentially many in $|\varphi| \in \Theta(d)$.

► **Example 6.** We give a protocol for $\varphi = (x - y \geq 2^d)$ with a polynomial number of states. This is essentially the protocol of [8]. We remove states and transitions from the protocol of Example 5, retaining only the states (ℓ, b, u) such that u is a power of 2, and some of the transitions involving these states:

$$\begin{aligned}
(*, *, 2^i), (*, *, 2^i) &\mapsto (A, N, 2^{i+1}), (P, N, 0) && \text{for every } 0 \leq i \leq d-2 \\
(*, *, 2^{d-1}), (*, *, 2^{d-1}) &\mapsto (A, Y, 2^d), (P, Y, 0) \\
(*, *, -2^i), (*, *, -2^i) &\mapsto (A, N, -2^{i+1}), (P, N, 0) && \text{for every } 0 \leq i \leq d-1 \\
(*, *, 2^i), (*, *, -2^i) &\mapsto (A, N, 0), (P, N, 0) && \text{for every } 0 \leq i \leq d-1
\end{aligned}$$

The protocol is not yet correct. For example, for $d = 1$ and the input $x = 2, y = 1$, the protocol can reach in one step the configuration in which the three agents (two x -agents and one y -agent) are in states $(A, Y, 2), (P, Y, 0), (A, N, -1)$, after which it gets stuck. In [8] this is solved by adding “reverse” transitions:

$$\begin{aligned}
(A, N, 2^{i+1}), (P, N, 0) &\mapsto (A, N, 2^i), (P, N, 2^i) && \text{for every } 0 \leq i \leq d-2 \\
(A, Y, 2^d), (P, Y, 0) &\mapsto (A, N, 2^{d-1}), (P, N, 2^{d-1}) \\
(A, N, -2^{i+1}), (P, N, 0) &\mapsto (A, N, -2^i), (A, N, -2^i) && \text{for every } 0 \leq i \leq d-1
\end{aligned}$$

The protocol has only $\Theta(d)$ states and transitions, but runs within $\Omega(n^{2^d-2})$ interactions. Consider the inputs x, y such that $x - y = 2^d$, and let $n := x + y$. Say that an agent is *positive* at a configuration if it has positive wealth at it. The protocol can only stabilise if it reaches a configuration with exactly one positive agent with wealth 2^d . Consider a configuration with $i < 2^d$ positive agents. The next configuration can have $i - 1, i$, or $i + 1$ positive agents. The probability of $i + 1$ positive agents is $\Omega(1/n)$, while that of $i - 1$ positive agents is only $\mathcal{O}(1/n^2)$, and the expected number of interactions needed to go from 2^d positive agents to only 1 is $\Omega(n^{2^d-1})$ [11, Appendix A.1].

► **Example 7.** Given protocols $\mathcal{P}_1, \mathcal{P}_2$ with n_1 and n_2 states deciding predicates φ_1 and φ_2 , Angluin et al. construct in [5] a protocol \mathcal{P} for $\varphi_1 \wedge \varphi_2$ with $n_1 \cdot n_2$ states. It follows that the number of states of a protocol for $\varphi := \varphi_1 \wedge \dots \wedge \varphi_s$ grows exponentially in s , and so in $|\varphi|$. Blondin et al. give an alternative construction with polynomially many states [8, Section 5.3]. However, their construction contains transitions that, as in the previous example, reverse the effect of other transitions, and make the protocol very slow. The problem is already observed in the toy protocol with states q_1, q_2 and transitions $q_1, q_1 \mapsto q_2, q_2$ and $q_1, q_2 \mapsto q_1, q_1$. (Similar transitions are used in the initialisation of [8].) Starting with an even number $n \geq 2$ of agents in q_1 , eventually all agents move to q_2 and stay there, but the expected number of interactions is $\Omega(2^{n/10})$ [11, Appendix A.2].

6 Succinct Bounded Population Computers for Presburger Predicates

In Sections 6.1 and 6.2 we construct population computers for remainder and threshold predicates in which all coefficients are powers of two. We present the remainder case in detail, and sketch the threshold case. The generalization to arbitrary coefficients is achieved by means of a gadget very similar to the one we used to compute boolean combinations of predicates. This later gadget is presented in Section 6.3, and so we introduce the generalization there.

6.1 Population computers for remainder predicates

Let $Pow^+ = \{2^i \mid i \geq 0\}$ be the set of positive powers of 2.

We construct population computers \mathcal{P}_φ for remainder predicates $\varphi := \sum_{i=1}^v a_i x_i \equiv_m c$, where $a_i \in Pow^+ \cap \{0, \dots, m-1\}$ for every $1 \leq i \leq v$, $m \in \mathbb{N}$, and $c \in \{0, \dots, m-1\}$. We say that a finite multiset r over Pow^+ *represents* the residue $\text{rep}(r) := \text{sum}(r) \bmod m$. For example, if $m = 11$ then $r_{18} := \{2^3, 2^3, 2^1\}$ represents 7. Accordingly, we call the

For $q = 2^d$ we introduce a transition that replaces an agent in 2^d by a multiset of agents r with $\text{sum}(r) = 2^d - m$, preserving the residue. Let $b_d b_{d-1} \dots b_0$ be the binary encoding of $2^d - m$, and let $\{i_1, \dots, i_j\}$ be the positions such that $b_{i_1} = \dots = b_{i_j} = 1$. The transition is:

$$2^d, 0, \dots, 0 \mapsto 2^{i_1}, \dots, 2^{i_j} \quad \langle \text{modulo} \rangle$$

These transitions are enough, but we also add a transition that takes d agents in 2^d and replaces them by agents with sum $d \cdot 2^d \bmod m$. Intuitively, this makes the protocol faster. Let $b_d b_{d-1} \dots b_0$ and $\{i_1, \dots, i_j\}$ be as above, but for $d \cdot 2^d \bmod m$ instead of $2^d - m$.

$$2^d, \dots, 2^d \mapsto 2^{i_1}, \dots, 2^{i_j}, 0, \dots, 0 \quad \langle \text{fast modulo} \rangle$$

Helpers. We set $H := \{3d \cdot 0\}$, i.e. the computer initially places at least $3d$ helper agents in state 0. This makes sure one can always execute the next $\langle \text{modulo} \rangle$ or $\langle \text{fast modulo} \rangle$ transition: if no more agents can be combined, there are at most d agents in the states $2^0, \dots, 2^{d-1}$. Thus, there are at least $2d$ agents in the states 0 and 2^d , enabling one of these transitions. Observe that for every initial configuration $C_I + C_H$ we have $\text{sum}(C_I + C_H) = \text{sum}(C_I)$, and so, abusing language, every initial configuration for C_I is also a representation of z .

Output function. The computer eventually reaches a support configuration with at most one agent in every state except for 0. Thus, for every support set $S \subseteq Q$, we define $O(S) := 1$ if $\text{sum}(S) \equiv_m c$, and $O(S) = 0$ else. We show the existence of a small boolean circuit for the output function O in the proof of Lemma 8; this can be found in [11, Appendix B.1].

► **Lemma 8.** *Let $\varphi := \sum_{i=1}^v a_i x_i \equiv_m c$, where $a_i \in \{2^{d-1}, \dots, 2^1, 2^0\}$ for every $1 \leq i \leq v$ and $c \in \{0, \dots, m-1\}$ with $d := \lceil \log_2 m \rceil$. There is a bounded computer of size $\mathcal{O}(d)$ deciding φ .*

The left half of Figure 1 shows the population computer for $\varphi = (8x + 5y \equiv_{11} 4)$.

6.2 Population computers for threshold predicates

We sketch the construction of population computers \mathcal{P}_φ for threshold predicates $\varphi := \sum_{i=1}^v a_i x_i \geq c$, where $a_i \in \{2^j, 2^{-j} \mid j \geq 0\}$ for every $1 \leq i \leq v$ and $c \in \mathbb{N}$. As the construction is similar to the construction for remainder, we will focus on the differences and refer to [11, Appendix B.2] for details.

As for remainder, we work with representations that are multisets of powers of 2. However, they represent the sum of their elements (without modulo) and we allow both positive and negative powers of 2. Similar to the remainder construction, the computer transforms any representation into a *support representation* without changing the represented value. Then, the computer decides the predicate using only the support of that representation.

Again, there are $\langle \text{combine} \rangle$ transitions that allow agents with the same value to combine. Instead of modulo transitions, $\langle \text{cancel} \rangle$ transitions further simplify the representation: $2^i, -2^i \mapsto 0, 0$. Note that even after exhaustively applying $\langle \text{combine} \rangle$ and $\langle \text{cancel} \rangle$ there can still be many agents in 2^d or many agents in -2^d . This has two consequences:

- In the construction for general predicates of Section 6.3, we need that computers for remainder and threshold move most agents to state 0. In the remainder construction, all but a constant number of agents are moved to 0. In contrast, the threshold construction does not have this property. Thus, we do not design a single computer for a given threshold predicate φ but a family: one for every degree d larger than some minimum degree $d_0 \in \Omega(|\varphi|)$. Intuitively, larger degrees result in a larger fraction of agents in 0.

11:10 Fast and Succinct Population Protocols for Presburger Arithmetic

■ Assume we detect agents in 2^d (-2^d is analogous). If there are many, the predicate is true. However, if there is just one, then the represented value might be small, due to negative contributions $-2^0, \dots, -2^{d-1}$. We cannot distinguish the two cases, so we add transition $\langle \text{cancel 2nd highest} \rangle: 2^d, -2^{d-1} \mapsto 2^{d-1}, 0$. It ensures that agents cannot be present in both 2^d and -2^{d-1} ; therefore, an agent in 2^d certifies a value of at least 2^{d-1} . The right half of Figure 1 shows the population computer for $\varphi = (-2x + y \geq 5)$ with degree $d = 4$. [11, Appendix B.2] proves:

► **Lemma 9.** *Let $\varphi := \sum_{i=1}^v a_i x_i \geq c$, where $a_i \in \{2^j, 2^{-j} \mid j \geq 0\}$ for every $1 \leq i \leq v$. For every $d \geq \max\{\lceil \log_2 c \rceil + 1, \lceil \log_2 |a_1| \rceil, \dots, \lceil \log_2 |a_v| \rceil\}$ there is a bounded computer of size $\mathcal{O}(d)$ that decides φ .*

6.3 Population computers for all Presburger predicates

We present a construction that, given threshold or remainder predicates $\varphi_1, \dots, \varphi_s$, yields a population computer \mathcal{P} deciding an arbitrary given boolean combination $B(\varphi_1, \dots, \varphi_s)$ of $\varphi_1, \dots, \varphi_s$. We only sketch the construction, see [11, Appendix B.3] for details. We use the example $\varphi_1 = (y - 2x \geq 5)$, $\varphi_2 = (8x + 5y \equiv_{11} 4)$ and $B(\varphi_1, \varphi_2) = \varphi_1 \vee \varphi_2$. The result of the construction for this example is shown in Figure 1. The construction has 6 steps:

1. Rewrite Predicates. The constructions in Sections 6.1 and 6.2 only work for predicates where all coefficients are powers of 2. We transform each predicate φ_i into a new predicate φ'_i where all coefficients are decomposed into their powers of 2. In our example, $\varphi'_1 := \varphi_1$ because all coefficients are already powers of 2. However, $\varphi_2(x, y) = (8x + 5y \equiv_{11} 4)$ is rewritten as $\varphi'_2(x, y_1, y_2) := (8x + 4y_1 + 1y_2 \equiv_{11} 4)$ because $5 = 4 + 1$. Note that $\varphi_2(x, y) = \varphi'_2(x, y, y)$ holds for every $x, y \in \mathbb{N}$. Let r be the size of the largest split of a coefficient, i.e. $r = 2$ in the example.

2. Construct Subcomputers. For every $1 \leq i \leq s$, if φ_i is a remainder predicate, then let \mathcal{P}_i be the computer defined in Section 6.1. If φ_i is a threshold predicate, then let \mathcal{P}_i be the computer of Section 6.2, with $d = d_0 + \lceil \log_2 s \rceil$. We explain this choice of d in step 5.

3. Combine Subcomputers. Take the disjoint union of \mathcal{P}_i , but merging their 0 states. More precisely, rename all states $q \in Q_i$ to $(q)_i$, with the exception of state 0. Construct a computer with the union of all the renamed states and transitions. Figure 1 shows the Petri net representation of the computer so obtained for our example. We call the combined 0 state *reservoir* as it holds agents with no value that are needed for various tasks like input distribution.

4. Input Distribution. For each variable x_i add a corresponding new input state x_i . Then add a transition that takes an agent in state x_i and agents in 0 and distributes agents to the input states of the subcomputers that correspond to x_i . In our example, we add two states x and y and the transitions $x, 0 \mapsto (1)_1, (8)_2$ and $y, 0, 0 \mapsto (-2)_1, (4)_2, (1)_2$. The distribution for x needs one helper, because we need one agent in each subcomputer. The distribution for y needs two helpers, one for \mathcal{P}_1 and two for \mathcal{P}_2 , as $5y$ was split into $4y_1 + 1y_2$. This way, once the input states are empty, the correct value is distributed to each subcomputer. Crucially, this input distribution can be fast as it is not reversible.

5. Add Extra Helpers. In addition to all helpers from the subcomputers, add $r - 1$ more helpers to state 0. Intuitively, this allows to distribute the first input agent. Because of our choice for d in threshold subcomputers, each subcomputer returns most agents back to state 0. More precisely, for each distribution the number of agents that do not get returned to 0 only increases by at most $\frac{1}{s}$ (per subcomputer). So in total only one agent is “consumed” per distribution and enough agents are returned to 0 for the next distribution to occur. In our example, the agents that stay in each of the $s = 2$ subcomputers only increases by at most $\frac{1}{2}$ per distribution. (In fact, remainder subcomputers return all distributed agents.)

6. Combine Output. Note that we can still decide φ_i from the support of the states in the corresponding subcomputer \mathcal{P}_i . We compute the output for φ by combining the outputs of the subcomputers $\mathcal{P}_1, \dots, \mathcal{P}_s$ according to $B(\varphi_1, \dots, \varphi_s)$. In our example, we set the output to 1 if and only if the output of \mathcal{P}_1 or \mathcal{P}_2 is 1.

In [11, Appendix B.3], we show that this computer is succinct, correct and bounded:

► **Theorem 1.** *For every predicate $\varphi \in QFPA$ there exists a bounded population computer of size $\mathcal{O}(|\varphi|)$ that decides φ .*

7 Converting Population Computers to Population Protocols

In this section we prove Theorem 2. We proceed in four steps, which must be carried out in the given order. Section 7.1 converts any bounded computer \mathcal{P} for $\text{double}(\varphi)$ of size m into a *binary* bounded computer \mathcal{P}_1 with $\mathcal{O}(m^2)$ states. Section 7.2 converts \mathcal{P}_1 into a binary bounded computer \mathcal{P}_2 with a *marked consensus output function* (a notion defined in the section). Section 7.3 converts \mathcal{P}_2 into a binary bounded computer \mathcal{P}_3 for φ — not $\text{double}(\varphi)$ — with a marked consensus output function *and no helpers*. Section 7.4 shows that \mathcal{P}_3 runs within $\mathcal{O}(n^3)$ interactions. Finally, we convert \mathcal{P}_3 to a binary terminating (not necessarily bounded) computer \mathcal{P}_4 with a *normal consensus output* and no helpers, also running within $\mathcal{O}(n^3)$ interactions. This uses standard ideas; for space reasons it is described only in the full version at [11, Appendix F]. Similarly, the other conversions and results are only sketched, with details in [11].

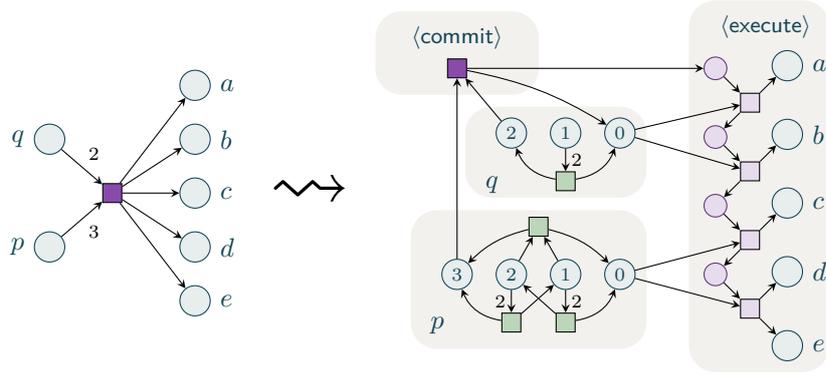
7.1 Removing multiway transitions

We transform a bounded population computer with k -way transitions $r \mapsto s$ such that $|\text{supp}(r)| \leq 2$ into a binary bounded population computer. Let us first explain why the construction introduced in [9, Lemma 3], which works for arbitrary transitions $r \mapsto s$, is too slow. In [9], the 3-way transition $t : q_1, q_2, q_3 \mapsto q'_1, q'_2, q'_3$ is simulated by the transitions

$$t_1 : q_1, q_2 \mapsto w, q_{12} \quad t_2 : q_{12}, q_3 \mapsto c_{12}, q'_3 \quad t_3 : q'_3, w \mapsto q'_1, q'_2 \quad \bar{t}_1 : w, q_{12} \mapsto q_1, q_2$$

Intuitively, the occurrence of t_1 indicates that two agents in q_1 and q_2 want to execute t , and are waiting for an agent in q_3 . If the agent arrives, then all three execute $t_2 t_3$, which takes them to q'_1, q'_2, q'_3 . Otherwise, the two agents must be able to return to q_1, q_2 to possibly execute other transitions. This is achieved by the “revert” transition \bar{t}_1 . The construction for a k -way transition has “revert” transitions $\bar{t}_1, \dots, \bar{t}_{k-2}$. As in Example 6 and Example 7, these transitions make the final protocol very slow.

We present a gadget without “revert” transitions that works for k -way transitions $r \mapsto s$ satisfying $|\text{supp}(r)| \leq 2$. Figure 2 illustrates it, using Petri net notation, for the 5-way transition $t : \{3p, 2q\} \mapsto \{a, b, c, d, e\}$. In the gadget, states p and q are split into $(p, 0), \dots, (p, 3)$



■ **Figure 2** Simulating the 5-way transition $\{3 \cdot p, 2 \cdot q \mapsto a, b, c, d, e\}$ by binary transitions.

and $(q, 0), \dots, (q, 2)$. Intuitively, an agent in (q, i) acts as representative for a group of i agents in state q . Agents in $(p, 3)$ and $(q, 2)$ commit to executing t by executing the binary transition $\langle \text{commit} \rangle$. After committing, they move to the states a, \dots, e together with the other members of the group, who are “waiting” in the states $(p, 0)$ and $(q, 0)$. Note that $\langle \text{commit} \rangle$ is binary because of the restriction $|\text{supp}(r)| \leq 2$ for multiway transitions.

To ensure correctness of the conversion, agents can commit to transitions if they represent more than the required amount. In this case, the initiating agents would commit to a transition and then elect representatives for the superfluous agents, before executing the transition. This requires additional intermediate states.

[11, Appendix C] formalises the gadget and proves its correctness and speed.

7.2 Converting output functions to marked-consensus output functions

We convert a computer with an arbitrary output function into another one with a *marked-consensus* output function. An output function is a *marked-consensus* output function if there are disjoint sets of states $Q_0, Q_1 \subseteq Q$ such that $O(S) := b$ if $S \cap Q_b \neq \emptyset$ and $S \cap Q_{1-b} = \emptyset$, for $b \in \{0, 1\}$, and $O(S) := \perp$ otherwise. Intuitively, for every $S \subseteq Q$ we have $O(S) = 1$ if all agents agree to avoid Q_0 (consensus), and at least one agent populates Q_1 (marked consensus). We only sketch the construction, a detailed description as well as a graphical example can be found in [11, Appendix D].

Our starting point is some bounded and binary computer $\mathcal{P} = (Q, \delta, I, O, H)$, e.g. as constructed in Section 7.1. Let (G, E) be a boolean circuit with only NAND-gates computing the output function O . We simulate \mathcal{P} by a computer \mathcal{P}' with a marked consensus output and $\mathcal{O}(|Q| + |G|)$ states. This result allows us to bound the number of states of \mathcal{P}' by applying well known results on the complexity of Boolean functions.

Intuitively, \mathcal{P}' consists of two processes running asynchronously in parallel. The first one is (essentially, see below) the computer \mathcal{P} itself. The second one is a gadget that simulates the execution of G on the support of the current configuration of \mathcal{P} . Whenever \mathcal{P} executes a transition, it raises a flag indicating that the gadget must be reset (for this, we duplicate each state $q \in Q$ into two states $(q, +)$ and $(q, -)$, indicating whether the flag is raised or lowered). Crucially, \mathcal{P} is bounded, and so it eventually performs a transition *for the last time*. This resets the gadget for the last time, after which the gadget simulates (G, E) on the support of the terminal configuration reached by \mathcal{P} .

The gadget is designed to be operated by one *state-helper* for each $q \in Q$, with set of states $Q_{\text{supp}}(q)$, and a *gate-helper* for each gate $g \in G$, with set of states $Q_{\text{gate}}(g)$, defined as follows:

- $Q_{\text{supp}}(q) := \{q\} \times \{0, 1, !\}$. These states indicate that q belongs/does not belong to the support of the current configuration (states $(q, 0)$ and $(q, 1)$), or that the output has changed from 0 to 1 (state $(q, !)$).
- $Q_{\text{gate}}(g) := \{g\} \times \{0, 1, \perp\}^3$ for each gate $g \in G$, storing the current values of the two inputs of the gate and its output. Uninitialised values are stored as \perp .

Recall that a population computer must also remain correct for a larger number of helpers. This is ensured by letting all helpers populating one of these sets, say $Q_{\text{supp}}(q)$, perform a leader election; whenever two helpers in states of $Q_{\text{supp}}(q)$ meet, one of them becomes a non-leader, and a flag requesting a complete reset of the gadget is raised. All resets are carried out by a *reset-helper* with set of states $Q_{\text{reset}} := \{0, \dots, |Q| + |G|\}$, initially in state 0. (Reset-helpers also carry out their own leader election!) Whenever a reset is triggered, the reset-helper contacts all other $|Q| + |G|$ helpers in round-robin fashion, asking them to reset the computation.

Eventually the original protocol \mathcal{P} has already reached a terminal configuration with some support Q_{term} , each set $Q_{\text{supp}}(q)$ and $Q_{\text{gate}}(g)$ is populated by exactly one helper, and all previous resets are terminated. From this moment on, \mathcal{P} never changes its configuration. The $|Q|$ state-helpers detect the support Q_{term} of the terminal configuration by means of transitions that move them to the states $Q_{\text{term}} \times \{1\}$ and $(Q \setminus Q_{\text{term}}) \times \{0\}$; the gate-helpers execute (G, E) on input Q' by means of transitions that move them to the states describing the correct inputs and outputs for each gate. State-helpers use $Q \times \{!\}$ as intermediate states, indicating that the circuit must recompute its output.

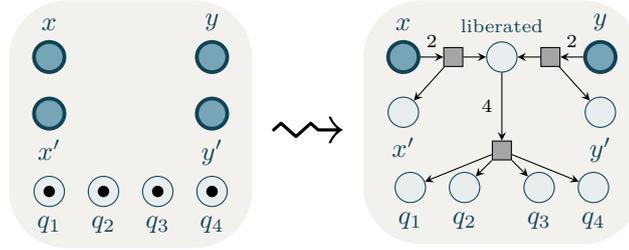
It remains to choose the sets Q_0 and Q_1 of states of the marked consensus output. We do it according to the output b of the output gate $g_{\text{out}} \in G$: Q_b is the set of states of $Q_{\text{gate}}(g_{\text{out}})$ corresponding to output b .

7.3 Removing helpers

We convert a bounded binary computer \mathcal{P} deciding the predicate $\text{double}(\varphi)$ over variables $x_1, \dots, x_k, x'_1, \dots, x'_k$ into a computer \mathcal{P}' with no helpers deciding φ over variables x_1, \dots, x_k . In [8], a protocol with helpers and set of states Q is converted into a protocol without helpers with states $Q \times Q$. We sketch a better construction that avoids the quadratic blowup. A detailed description can be found in [11, Appendix E].

Let us give some intuition first. All agents of an initial configuration of \mathcal{P}' are in input states. \mathcal{P}' simulates \mathcal{P} by *liberating* some of these agents and transforming them into helpers, without changing the output of the computation. For this, two agents in an input state x_i are allowed to interact, producing one agent in x'_i and one “liberated” agent, which can be used as a helper. This does not change the output of the computation, because $\text{double}(\varphi)(\dots, x_i, \dots, x'_i, \dots) = \text{double}(\varphi)(\dots, x_i - 2, \dots, x'_i + 1, \dots)$ holds by definition of $\text{double}(\varphi)$.

Figure 3 illustrates this idea. Assume \mathcal{P} has input states x, y, x', y' and helpers $H = \{q_1, q_2, q_3, q_4\}$, as shown on the left-hand side. Assume further that \mathcal{P} computes a predicate $\text{double}(\varphi)(x, y, x', y')$. The computer \mathcal{P}' is shown on the right of the figure. The additional transitions liberate agents, and send them to the helper states H . Observe that the initial states of \mathcal{P}' are only x and y . Let us see why \mathcal{P}' decides $\varphi(x, y)$. As the initial configuration of



■ **Figure 3** Illustration in graphical Petri net notation (see Section 3) of construction that removes helpers. Initial states are highlighted.

\mathcal{P}' for an input x, y puts no agents in x', y' , the computer \mathcal{P}' produces the same output on input x, y as \mathcal{P} on input $x, y, 0, 0$. Since \mathcal{P} decides $\text{double}(\varphi)$ and $\text{double}(\varphi)(x, y, 0, 0) = \varphi(x, y)$ by the definition of $\text{double}(\varphi)$, we are done. We make some remarks:

- \mathcal{P}' may liberate more agents than necessary to simulate the multiset H of helpers of \mathcal{P} . This is not an issue, because by definition additional helpers do not change the output of the computation.
- If the input is too small, \mathcal{P}' cannot liberate enough agents to simulate H . Therefore, the new computer only works for inputs of size $\Omega(|H|) = \Omega(|\varphi|)$.
- Even if the input is large enough, \mathcal{P}' might move agents out of input states before liberating enough helpers. However, the computers of Section 6 can only do this if there are enough helpers in the reservoir state (see point 3. in Section 6.3). Therefore, they always generate enough helpers when the input is large enough.

7.4 A $\mathcal{O}(n^3)$ bound on the expected interactions

We show that the computer obtained after the previous conversion runs within $\mathcal{O}(n^3)$ interactions. We sketch the main ideas; the details are in [11, Appendix G].

We introduce *potential functions* that assign to every configuration a positive *potential*, with the property that executing any transition strictly decreases the potential. Intuitively, every transition “makes progres”. We then prove two results: (1) under a mild condition, a computer has a potential function iff it is bounded, and (2) every binary computer with a potential function and no helpers, i.e. any bounded computer for which speed is defined, stabilises within $\mathcal{O}(n^3)$ interactions. This concludes the proof.

Fix a population computer $\mathcal{P} = (Q, \delta, I, O, H)$.

► **Definition 10.** A function $\Phi : \mathbb{N}^Q \rightarrow \mathbb{N}$ is linear if there exist weights $w : Q \rightarrow \mathbb{N}$ s.t. $\Phi(C) = \sum_{q \in Q} w(q)C(q)$ for every $C \in \mathbb{N}^Q$. We write $\Phi(q)$ instead of $w(q)$. A potential function (for \mathcal{P}) is a linear function Φ such that $\Phi(r) \geq \Phi(s) + |r| - 1$ for all $(r \mapsto s) \in \delta$.

Observe that k -way transitions reduce the potential by $k - 1$, binary transitions by 1. At this point, we consider only binary computers, but this distinction becomes relevant for the refined speed analysis.

If a population computer has a potential function, then every run executes at most $\mathcal{O}(n)$ transitions, and so the computer is bounded. Applying Farkas’ Lemma we can show that the converse holds for computers in which every state can be populated – a mild condition, since states that can never be populated can be deleted without changing the behaviour.

► **Lemma 11.** If \mathcal{P} has a reachable configuration C_q with $C_q(q) > 0$ for each $q \in Q$, then \mathcal{P} is bounded iff there is a potential function for \mathcal{P} .

Consider now a binary computer with a potential function and no helpers. At every non-terminal configuration, at least one (binary) transition is enabled. The probability that two agents chosen uniformly at random enable this transition is $\Omega(1/n^2)$, and so a transition occurs within $\mathcal{O}(n^2)$ interactions. Since the computer has a potential function, every run executes at most $\mathcal{O}(n)$ transitions, and so the computer stabilises within $\mathcal{O}(n^3)$ interactions.

The final step to produce a population protocol is to translate computers with marked-consensus output function into computers with standard consensus output function, while preserving the number of interactions. For space reasons this construction is presented in [11, Appendix F].

8 Rapid Population Computers: Proving a $\mathcal{O}(n^2)$ Bound

We refine our running-time analysis to show that the population protocols we have constructed actually stabilise within $\mathcal{O}(n^2)$ interactions. We continue to use potential functions, as introduced in Section 7.4, but improve our analysis as follows:

- We introduce *rapidly-decreasing* potential functions. Intuitively, their existence shows that progress is not only *possible*, but also *likely*. We prove that they certify stabilisation within $\mathcal{O}(n^2)$ interactions.
- We introduce *rapid* population computers, as computers with rapidly-decreasing potential functions that also satisfy some technical conditions. We convert rapid computers into protocols with $\mathcal{O}(|\varphi|)$ states, and show that the computers of Section 6 are rapid.

In order to define rapidly-decreasing potential functions, we need a notion of “probability to execute a transition” that generalises to multiway transitions and is preserved by our conversions. At a configuration C of a protocol, the probability of executing a binary transition $t = (p, q \mapsto p', q')$ is $C(q)C(p)/n(n-1)$. Intuitively, leaving out the normalisation factor $1/n(n-1)$, the transition has “speed” $C(q)C(p)$, proportional in the *product* of the number of agents in p and q . But for a multiway transition like $q, q, p \mapsto r_1, r_2, r_3$ the situation changes. If $C(q) = 2$, it does not matter how many agents are in p – the transition is always going to take $\Omega(n^2)$ interactions. We therefore define the speed of a transition as $\min\{C(q), C(p)\}^2$ instead of $C(q)C(p)$.

For the remainder of this section, let $\mathcal{P} = (Q, \delta, I, O, H)$ denote a population computer.

► **Definition 12.** *Given a configuration $C \in \mathbb{N}^Q$ and some transition $t = (r \mapsto s) \in \delta$, we let $\text{tmin}_t(C) := \min\{C(q) : q \in \text{supp}(r)\}$. For a set of transitions $T \subseteq \delta$, we define $\text{speed}_T(C) := \sum_{t \in T} \text{tmin}_t(C)^2$, and write $\text{speed}(C) := \text{speed}_\delta(C)$ for convenience.*

► **Definition 13.** *Let Φ denote a potential function for \mathcal{P} and let $\alpha \geq 1$. We say that Φ is α -rapidly decreasing at a configuration C if $\text{speed}(C) \geq (\Phi(C) - \Phi(C_{\text{term}}))^2/\alpha$ for all terminal configurations C_{term} with $C \rightarrow C_{\text{term}}$.*

We have not been able to find potential functions for the computers of Section 6 that are rapidly decreasing at every reachable configuration, only at reachable configurations with sufficiently many helpers, defined below. Fortunately, that is enough for our purposes.

► **Definition 14.** *$C \in \mathbb{N}^Q$ is well-initialised if C is reachable and $C(I) + |H| \leq \frac{2}{3}n$.*

Observe that an initial configuration C can only be well-initialised if $C(\text{supp}(H)) \in \Omega(C(I))$. We now define *rapid* population computers, and state the result of our improved analysis.

► **Definition 15.** \mathcal{P} is α -rapid if

1. it has a potential function Φ which is α -rapidly decreasing in well-initialised configurations,
2. every state of \mathcal{P} but one has at most 2 outgoing transitions,
3. all configurations in \mathbb{N}^I are terminal, and
4. for all transitions $t = (r \mapsto s)$, $q \in I$ we have $r(q) \leq 1$ and $s(q) = 0$.

► **Theorem 3.**

- (a) The population computers constructed in Theorem 1 are $\mathcal{O}(|\varphi|^3)$ -rapid.
- (b) Every α -rapid population computer of size m deciding $\text{double}(\varphi)$ can be converted into a terminating population protocol with $\mathcal{O}(m)$ states that decides φ in $\mathcal{O}(\alpha m^4 n^2)$ interactions for inputs of size $\Omega(m)$.

The detailed proofs can be found in the full version [11], in the following sections. The proof of (a) is given in Appendix B. For (b), we prove separate theorems for each conversion in Appendices C, D, E, and F. To achieve a tighter analysis of our conversions, we generalise the notion of potential function; this is described in Appendix H.

9 Conclusions

We have shown that every predicate φ of quantifier-free Presburger arithmetic has a population protocol with $\mathcal{O}(\text{poly}(|\varphi|))$ states and $\mathcal{O}(|\varphi|^7 \cdot n^2)$ expected number of interactions. If only inputs of size $\Omega(|\varphi|)$ matter, we give a protocol with $\mathcal{O}(|\varphi|)$ states and the same speed. The obvious point for further improvement is the $|\varphi|^7$ factor in the expected number of interactions.

Our construction is close to optimal. Indeed, for every construction there is an infinite family of predicates for which it yields protocols with $\Omega(|\varphi|^{1/4})$ states [9]; further, it is known that every protocol for the majority predicate requires in $\Omega(n^2)$ interactions.

References

- 1 Dan Alistarh, James Aspnes, David Eisenstat, Rati Gelashvili, and Ronald L. Rivest. Time-space trade-offs in population protocols. In *SODA*, pages 2560–2579. SIAM, 2017.
- 2 Dan Alistarh and Rati Gelashvili. Recent algorithmic advances in population protocols. *SIGACT News*, 49(3):63–73, 2018.
- 3 Dan Alistarh, Rati Gelashvili, and Milan Vojnovic. Fast and exact majority in population protocols. In *PODC*, pages 47–56. ACM, 2015.
- 4 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. In *PODC*, pages 290–299. ACM, 2004.
- 5 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Comput.*, 18(4):235–253, 2006.
- 6 Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. *Distributed Comput.*, 21(3):183–199, 2008.
- 7 Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Comput.*, 20(4):279–304, 2007.
- 8 Michael Blondin, Javier Esparza, Blaise Genest, Martin Helfrich, and Stefan Jaax. Succinct population protocols for Presburger arithmetic. In *STACS*, volume 154 of *LIPICs*, pages 40:1–40:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
- 9 Michael Blondin, Javier Esparza, and Stefan Jaax. Large flocks of small birds: On the minimal size of population protocols. In *STACS*, volume 96 of *LIPICs*, pages 16:1–16:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

- 10 Robert Brijder, David Doty, and David Soloveichik. Democratic, existential, and consensus-based output conventions in stable computation by chemical reaction networks. *Natural Computing*, 17(1):97–108, 2018.
- 11 Philipp Czerner, Roland Guttenberg, Martin Helfrich, and Javier Esparza. Fast and succinct population protocols for Presburger arithmetic, 2022. [arXiv:2202.11601v2](https://arxiv.org/abs/2202.11601v2).
- 12 David Doty, Mahsa Eftekhari, Leszek Gasieniec, Eric E. Severson, Grzegorz Stachowiak, and Przemyslaw Uznanski. Brief announcement: A time and space optimal stable population protocol solving exact majority. In *PODC*, pages 77–80. ACM, 2021.
- 13 Robert Elsässer and Tomasz Radzik. Recent results in population protocols for exact majority and leader election. *Bull. EATCS*, 126, 2018.
- 14 Christoph Haase. A survival guide to Presburger arithmetic. *ACM SIGLOG News*, 5(3):67–82, 2018.

Local Mutual Exclusion for Dynamic, Anonymous, Bounded Memory Message Passing Systems

Joshua J. Daymude  

Biodesign Center for Biocomputing, Security and Society,
Arizona State University, Tempe, AZ, USA

Andréa W. Richa  

School of Computing and Augmented Intelligence,
Arizona State University, Tempe, AZ, USA

Christian Scheideler  

Department of Computer Science, Universität Paderborn, Germany

Abstract

Mutual exclusion is a classical problem in distributed computing that provides *isolation* among concurrent action executions that may require access to the same shared resources. Inspired by algorithmic research on distributed systems of weakly capable entities whose connections change over time, we address the *local mutual exclusion* problem that tasks each node with acquiring exclusive locks for itself and the maximal subset of its “persistent” neighbors that remain connected to it over the time interval of the lock request. Using the established *time-varying graphs* model to capture adversarial topological changes, we propose and rigorously analyze a local mutual exclusion algorithm for nodes that are anonymous and communicate via asynchronous message passing. The algorithm satisfies *mutual exclusion* (non-intersecting lock sets) and *lockout freedom* (eventual success with probability 1) under both semi-synchronous and asynchronous concurrency. It requires $\mathcal{O}(\Delta)$ memory per node and messages of size $\Theta(1)$, where Δ is the maximum number of connections per node. We conclude by describing how our algorithm can implement the pairwise interactions assumed by *population protocols* and the concurrency control operations assumed by the *canonical amoebot model*, demonstrating its utility in both passively and actively dynamic distributed systems.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Concurrency; Software and its engineering → Mutual exclusion

Keywords and phrases Mutual exclusion, dynamic networks, message passing, concurrency

Digital Object Identifier 10.4230/LIPIcs.SAND.2022.12

Related Version *Full Version*: <https://arxiv.org/abs/2111.09449>

Funding *Joshua J. Daymude*: NSF (CCF-1733680), U.S. ARO (MURI W911NF-19-1-0233), the Momental Foundation’s Mistletoe Research Fellowship, and the ASU Biodesign Institute.

Andréa W. Richa: NSF (CCF-1733680, CCF-2106917) and U.S. ARO (MURI W911NF-19-1-0233).

Christian Scheideler: DFG Project SCHE 1592/6-1.

1 Introduction

Distributed computing research has grown increasingly concerned with characterizing the capabilities and limitations of systems composed of *dynamic entities* (or *nodes*). Recently, these studies have considered both biological collectives such as social insects [2, 14, 26], spiking neural networks [49], and DNA and molecular computers [13, 39, 51] as well as engineered systems such as overlay networks and the Internet of Things (IoT) [23], swarm and modular self-reconfigurable robotics [25, 28, 40, 53], and programmable matter [3, 17, 19, 37]. Entities in these systems often make decisions based only on their own knowledge (or “state”), locally-perceptible measures of their environment (e.g., pheromones, the number or density of nearby neighbors, etc.), and information communicated to them by their neighbors.



© Joshua J. Daymude, Andréa W. Richa, and Christian Scheideler;
licensed under Creative Commons License CC-BY 4.0

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).

Editors: James Aspnes and Othon Michail; Article No. 12; pp. 12:1–12:19

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Compared to the static setting where acting nodes’ neighborhoods do not change, designing correct distributed algorithms in the dynamic setting is a challenging task. In this paper, we use the established *time-varying graphs* (TVGs) model [11, 12] to capture adversarial changes in network topology and consider weakly capable nodes that are *anonymous*, have *bounded memory*, communicate via *asynchronous message passing*, and execute their algorithms *semi-synchronously* or *asynchronously*. The classical *mutual exclusion* problem [20] regulates how nodes enter their critical sections using locks, defined as a pair of operations LOCK and UNLOCK. Our *local mutual exclusion* problem – designed to enable nodes to locally coordinate their interactions in the dynamic, concurrent setting – defines LOCK as a node acquiring locks for itself and the maximal subset of its “persistent” neighbors that remain connected to it while the request is processed. A core challenge in designing such a LOCK operation in the dynamic setting lies in the nodes’ inability to know, when issuing lock requests, which neighbors will be persistent and which others will later be removed.

This locking mechanism greatly simplifies the design of local distributed algorithms in highly dynamic settings by providing *isolation* for concurrently executed actions. An algorithm’s actions can first be designed for the simpler sequential setting in which at most one node is active (potentially changing the system configuration) at a time. When considering the concurrent setting, each action is then treated as a critical section wrapped in a LOCK/UNLOCK pair; this ensures that no two simultaneously executing actions can involve overlapping neighborhoods. Our locking mechanism gracefully handles neighbor disconnections, ensuring that the locked and connected subset of an acting node’s neighborhood remains fixed throughout the execution of its action, just as it would be in the sequential setting. Thus, our locking mechanism restricts the algorithm designer’s concern from all possible complications arising from concurrent dynamics to just one: New connections may concurrently be established with a node while it is executing an action.

Our Contributions. We summarize our contributions as follows.

- A formalization of the *local mutual exclusion problem* in an extension of the time-varying graphs model that captures topological changes, asynchronous message passing, and semi-synchronous or asynchronous node activation (Section 2).
- A randomized algorithm implementing the LOCK and UNLOCK operations for local mutual exclusion that satisfies *mutual exclusion* (non-intersecting lock sets) and *lockout freedom* (eventual success with probability 1) under both semi-synchronous and asynchronous concurrency. This algorithm requires $\mathcal{O}(\Delta)$ memory per node and messages of size $\Theta(1)$, where Δ is the maximum number of connections per node (Sections 3–5).
- Applications of this local mutual exclusion algorithm to *population protocols* [3], establishing an underlying mechanism for guaranteeing pairwise interactions in a broader class of concurrent activation models, and the *canonical amoebot model* [17], implementing the model’s concurrency control operations (Section 6).

Related Work

Designing algorithms for concurrent computing environments is a challenging task requiring the careful control of simultaneously interacting processes and coordinated access to shared resources. Since its introduction by Dijkstra [20], the closely related *mutual exclusion* problem has received much attention from the research community. For shared memory systems, mutual exclusion can be conveniently solved by atomic operations like compare-and-swap, test-and-set, and fetch-and-add [29]. In contrast, our present focus is on asynchronous message passing. Classical approaches to mutual exclusion in asynchronous message passing

systems often assume that nodes have unique identifiers and global coordination (see, e.g., the survey [47]) or make use of unbounded counters like Lamport clocks (e.g., [35, 43]), neither of which are appropriate for the anonymous, bounded memory nodes we consider here. The most relevant classical algorithm to our setting is the *arrow protocol* [18, 41] that requires only constant memory per node to locally maintain a spanning tree rooted at the node with exclusive access to the shared resource; however, despite recent improvements [27, 32], it is not clear how to adapt this protocol to systems with dynamic topologies.

Our local variant of the mutual exclusion problem blurs the usual delineation between processes and the shared resources they're accessing as nodes compete to gain exclusive access to their neighborhoods. Like the well-studied *k-mutual exclusion* [24] and *group mutual exclusion* [30, 31] variants, ours allows multiple nodes to be in their critical sections simultaneously; however, these variants allow multiple process to access the same shared resource(s) concurrently while ours requires that concurrently locked neighborhoods be non-intersecting. This constraint is similar to ensuring the active nodes form a *distance-3 independent set* from graph theory and is related to the more general (α, β) -*ruling sets* [5] recently solved under the LOCAL and CONGEST models [33, 44]; however, these distributed algorithms rely on static topologies, unique identifiers, and synchronous message delivery. The recent results on mutual exclusion for *fully anonymous* systems [42], like our nodes and their neighborhoods, assume that neither the processes nor the shared resources have unique identifiers. However, like the earlier classical results above and other recent models of weak finite automata [21, 22], these do not extend to dynamic network topologies.

Research on *mobile ad hoc networks* (MANETs) directly embraces node and edge dynamics, modeling wireless communication links that form and fail as nodes move in and out of each other's transmission radii. Mutual exclusion has been exhaustively studied under MANET models [4, 6, 7, 15, 45, 50], and many of those ideas inspired recent work on mutual exclusion for intersection traffic control for autonomous vehicles [46, 52]. Mutual exclusion for MANETs is almost always solved using a token-based approach, sometimes combined with the imposition of a logical structure like a ring or tree. These approaches only apply to competitions for a single shared resource or critical section per token type; our nodes' competitions over their local neighborhoods would need one token type per neighboring node which is not addressed by prior work. More relevant to our local variant of mutual exclusion are randomized backoff mechanisms for local contention resolution used by MANETs and wireless networks [8, 9, 10] to ensure no two nodes are broadcasting in overlapping neighborhoods; however, these rely on nodes' chosen backoff delays to correspond to a consistent wall clock that is incompatible with our weaker model of concurrency. In any case, the standard MANET communication model of wireless broadcast with time-ordered, instantaneous receipt of messages is more powerful than our asynchronous message passing. Like MANETs, algorithms for *self-stabilizing overlay networks* (see [23] for a recent survey) similarly embrace node and edge dynamics, but often use more memory than our present algorithm and assume unique node identifiers.

Finally, we briefly highlight related models of *dynamic networks*, i.e., those whose structural properties change over time. Our model is closely related to the *time-varying graphs* (TVGs) model [11, 12] which unifies prior models of dynamic networks by capturing graph structural evolution over time through adversarial dynamics. We join recent work on message passing algorithms for TVGs that address the challenge of rapidly changing network topology [1]. The local nature of our mutual exclusion problem enables us to weaken the assumptions considered by prior works in this area. For example, we allow messages to have arbitrary but finite delays akin to asynchronous message passing, we assume weaker "semi-synchronous" and asynchronous models of concurrency, and we trade globally unique node

identifiers for local port labels. In Section 6, we demonstrate how the rapid dynamics modeled by TVGs combined with these weak assumptions on node capabilities facilitate application of our mutual exclusion algorithm to both systems with *passive* dynamics [3, 48], in which nodes have no control over topological changes, and those with *active* dynamics [17, 36, 38] in which nodes control the connections they establish and sever (e.g., via movements).

2 Preliminaries

2.1 Computational Model

We consider a distributed system composed of a fixed set of nodes V . Each node is assumed to be *anonymous*, lacking a unique identifier, and has a local memory storing its *state*. Nodes communicate with each other via message passing over a communication graph whose topology changes over time. We model this topology using a *time-varying graph* $\mathcal{G} = (V, E, T, \rho)$ where V is the set of nodes, E is a (static) set of undirected pairwise edges between nodes, $T = \{0, \dots, t_{\max}\}$ for some (possibly infinite) $t_{\max} \in \mathbb{N}$ is called the *lifetime* of \mathcal{G} , and $\rho : E \times T \rightarrow \{0, 1\}$ is the *presence* function indicating whether or not a given edge exists at a given time. A *snapshot* of \mathcal{G} at time $t \in T$ is the undirected graph $G_t = (V, \{e \in E : \rho(e, t) = 1\})$ and the *neighborhood* of a node $u \in V$ at time $t \in T$ is the set $N_t(u) = \{v \in V : \rho(\{u, v\}, t) = 1\}$. For $i \geq 0$, the i -th *round* lasts from time i to the instant just before time $i + 1$; thus, the communication graph in round i is G_i .

We assume that an adversary controls the presence function ρ and that E is the complete set of edges on nodes V ; i.e., we do not limit which edges the adversary can introduce. The only constraint we place on the adversary's topological changes is $\forall t \in T, u \in V, |N_t(u)| \leq \Delta$, where $\Delta > 0$ is the fixed number of *ports* per node. When the adversary establishes a new connection between nodes u and v , it must assign the endpoints of edge $\{u, v\}$ to open ports on u and v (and cannot do so if either node has no open ports). Node u locally identifies $\{u, v\}$ using its corresponding *port label* $\ell \in \{1, \dots, \Delta\}$ and v does likewise. For convenience of notation, we use $\ell_u(v)$ to refer to the label of the port on node u that is assigned to the edge $\{u, v\}$; this mapping of port labels to nodes is not available to the nodes. Edge endpoints remain in their assigned ports (and thus labels remain fixed) until disconnection, but nodes u and v may label $\{u, v\}$ differently and their labels are not known to each other a priori. Each node has a *disconnection detector* that adds the label of any port whose connection is severed to a set $D \subseteq \{1, \dots, \Delta\}$. A node's disconnection detector provides it a snapshot of D whenever it starts an action execution (see below) and then resets D to \emptyset .¹

Nodes communicate via message passing. A node u sends a message m to a neighbor v by calling $\text{SEND}(m, \ell_u(v))$. Message m remains in transit until either v receives and processes m at a later round chosen by the adversary, or u and v are disconnected and m is lost. Multiple messages in transit from u to v may be received by v in a different order than they were sent. A node always knows from which port it received a given message.

All nodes execute the same distributed algorithm \mathcal{A} , which is a set of *actions* each of the form $\langle \text{label} \rangle : \langle \text{guard} \rangle \rightarrow \langle \text{operations} \rangle$. An action's *label* specifies its name. Its *guard* is a Boolean predicate determining whether a node u can execute it based on the state of u and any message in transit that u may receive. An action is *enabled* for a node u if its guard is TRUE for u ; a node u is *enabled* if it has at least one enabled action. An action's

¹ Without this assumption, the adversary could disconnect an edge assigned to port ℓ of node u and then immediately connect a different edge to ℓ , causing an indistinguishability issue for node u .

operations specify what a node does when executing the action, structured as (i) receiving at most one message chosen by the adversary, (ii) a finite amount of internal computation and state updates, and (iii) at most one call to $\text{SEND}(m, \ell)$ per port label ℓ .

Each node executes its own instance of \mathcal{A} independently, sequentially (executing at most one action at a time), and reliably (meaning we do not consider crash or Byzantine faults). We assume an adversary controls the timing of node activations and action executions. When the adversary activates a node, it also chooses exactly one of the node's enabled actions for the node to execute; we note that this choice must be compatible with any message the adversary chooses to deliver to the node. In this work, we primarily focus on *semi-synchronous* activations, which we interpret in the time-varying graph context to mean that in each round, the adversary activates any (possibly empty) subset of enabled nodes concurrently and the activated nodes execute their specified actions within that round. In Section 5, we additionally consider *asynchronous* activations in which action executions may span arbitrary finite time intervals. We only constrain the adversary by *weak fairness*, meaning it must activate nodes such that any continuously enabled action is eventually executed and any message in transit on a continuously existent edge is eventually processed.

2.2 Local Mutual Exclusion

In the classical problem of *mutual exclusion*, nodes enter their critical sections using locks, defined as a pair of operations LOCK and UNLOCK (or “acquire” and “release”). A node issues a lock request by calling LOCK; once acquired, it is assumed that a node eventually releases these locks by calling UNLOCK. Our *local mutual exclusion* variant is concerned with nodes acquiring exclusive access to themselves and their immediate neighbors, though in the present context of dynamic networks, these neighborhoods may change over time.

Formally, each node u stores a variable $\text{lock} \in \{\perp, 0, \dots, \Delta\}$ that is equal to \perp if u is unlocked, 0 if u has locked itself, and $\ell_u(v) \in \{1, \dots, \Delta\}$ if u is locked by v . The *lock set* of a node u in round i is $\mathcal{L}_i(u) = \{v \in N_i(u) : \text{lock}(v) = \ell_v(u)\}$ which additionally includes u itself if $\text{lock}(u) = 0$. Suppose that in round i , a node u calls LOCK to issue a lock request of its current closed neighborhood $N_i[u] = \{u\} \cup N_i(u)$. This lock request *succeeds* at some later round $j > i$ if round j is the first in which $\mathcal{L}_j(u) = \{u\} \cup \{v \in N_i(u) : \forall t \in [i, j], \{u, v\} \in G_t\}$; i.e., j is the earliest round in which u obtains locks for itself and every *persistent* neighbor that remained connected to u in rounds i through j . Our goal is to design an algorithm \mathcal{A} implementing LOCK and UNLOCK that satisfies the following properties:

- *Mutual Exclusion.* For all rounds $i \in T$ and all pairs of nodes $u, v \in V$, $\mathcal{L}_i(u) \cap \mathcal{L}_i(v) = \emptyset$.
- *Lockout Freedom.* Every issued lock request eventually succeeds with probability 1.

Following the long tradition of mutual exclusion problem definitions, our local mutual exclusion problem is defined in terms of mutual exclusion and fairness properties. However, because each lock variable points to at most one node per time, it is impossible for two nodes' lock sets to intersect, trivially satisfying the mutual exclusion property. Nevertheless, satisfying lockout freedom remains challenging, especially in highly dynamic settings. When issuing lock requests, nodes do not know which of their connections will remain stable and which will disconnect by the time their coordination is complete. Thus, our problem variant captures what it means for nodes to lock their maximal persistent neighborhoods despite unpredictable and rapid topological changes.

■ **Table 1** The notation, domain, initialization, and description of the local variables used in the algorithm for local mutual exclusion by a node u .

Var.	Domain	Init.	Description
lock	$\{\perp, 0, \dots, \Delta\}$	\perp	\perp if u is unlocked, 0 if u has locked itself, and $\ell_u(v)$ if u is locked by v
state	$\{\perp, \text{PREPARE}, \text{COMPETE}, \text{WIN}, \text{LOCKED}, \text{UNLOCK}\}$	\perp	The lock state of node u
phase	$\{\perp, \text{PREPARE}, \text{COMPETE}\}$	\perp	The algorithm phase node u is in
L	$\subseteq N[u]$	\emptyset	Ports (nodes) u intends to lock
R	$\subseteq N[u]$	\emptyset	Ports via which u has received <code>ready()</code> , <code>ack-lock()</code> , or <code>ack-unlock()</code> responses
W	$\subseteq N[u] \times \{\text{TRUE}, \text{FALSE}\}$	\emptyset	Port-outcome pairs of <code>win()</code> messages u has received
H	$\subseteq N[u]$	\emptyset	Ports (nodes) on hold for the competition to lock u
A	$\subseteq N[u]$	\emptyset	Ports (nodes) of applicants that can join the competition to lock u
C	$\subseteq N[u]$	\emptyset	Ports (nodes) of candidates competing to lock u
P	$\subseteq C(u) \times \{0, \dots, K-1\}$	\emptyset	Port-priority pairs of the candidates

3 Algorithm for Local Mutual Exclusion

Our randomized algorithm for the local mutual exclusion problem specifies actions for the execution of LOCK and UNLOCK operations satisfying mutual exclusion and lockout freedom. An execution of the LOCK operation by a node u is organized into two phases: a *preparation phase* (Algorithm 1) in which u determines and notifies the nodes $L(u)$ it intends to lock, and a *competition phase* (Algorithm 2) in which u attempts to lock all nodes in $L(u)$, contending with any other nodes v for which $L(v) \cap L(u) \neq \emptyset$. An execution of the UNLOCK operation (Algorithm 3) by node u is straightforward, simply notifying all nodes in $L(u)$ that their locks are released. All local variables used in our algorithm are listed in Table 1 as they appear in the pseudocode. In a slight abuse of notation, we use $N[u]$ and the subsets thereof to represent both the nodes in the closed neighborhood of u and the port labels of u they are connected to. For clarity of presentation, the algorithm pseudocode allows for a node to send messages to itself (via “port 0”) just as it sends messages to its neighbors, though in reality these self-messages would be implemented with in-memory variable updates.

We refer to nodes that call LOCK/UNLOCK as *initiators* and the nodes that are being locked or unlocked as *participants*; it is possible for a node to be an initiator and participant simultaneously. Initiators progress through a series of *lock states* associated with the `state` variable; participants advance through the algorithm’s *phases* as indicated by the `phase` variable. We first describe the algorithm from an initiator’s perspective and then describe the complementary participants’ actions. A special CLEANUP helper function ensures that the nodes adapt to any disconnections affecting their variables that may have occurred since they last acted, so we omit the handling of these disconnections in the following description.

When an initiator u calls LOCK, it advances to the PREPARE state, sets $L(u)$ to all nodes in its closed neighborhood $N[u]$, and then sends `prepare()` messages to all nodes of $L(u)$. Once it has received `ready()` responses from all nodes of $L(u)$, it advances to the COMPETE state and joins the competitions for each node in $L(u)$ by sending `request-lock(p)` messages to all nodes of $L(u)$, where p is a priority chosen uniformly at random from $\{0, \dots, K-1\}$ for a fixed $K = \Theta(1)$. It then waits for the outcomes of these competitions. If it receives at

■ **Algorithm 1** The LOCK Operation: Preparation Phase for Node u .

```

1: INITLOCK: On LOCK being called  $\rightarrow$  ▷ Initiator initiates a lock request.
2:   if  $\text{state} = \perp$  then ▷ Only one locking operation at a time.
3:     CLEANUP().
4:     Set  $\text{state} \leftarrow \text{PREPARE}$  and  $L \leftarrow N[u]$ .
5:     for all  $\ell \in L$  do SEND( $\text{prepare}()$ ,  $\ell$ ).
6: RECEIVEPREPARE: On receiving  $\text{prepare}()$  via port  $\ell \rightarrow$ 
7:   CLEANUP().
8:   if  $\text{phase} = \text{COMPETE}$  then set  $H \leftarrow H \cup \{\ell\}$ . ▷ Put  $\ell$  on hold if already competing.
9:   else
10:    Set  $A \leftarrow A \cup \{\ell\}$  and  $\text{phase} \leftarrow \text{PREPARE}$ . ▷ Add  $\ell$  as an applicant otherwise.
11:    SEND( $\text{ready}()$ ,  $\ell$ ).
12: RECEIVEREADY: On receiving  $\text{ready}()$  via port  $\ell \rightarrow$ 
13:   CLEANUP().
14:   Set  $R \leftarrow R \cup \{\ell\}$ .
15: CHECKSTART:  $(\text{state} = \text{PREPARE}) \wedge (R = L) \rightarrow$  ▷ All  $\text{ready}()$  messages received.
16:   CLEANUP().
17:   Set  $\text{state} \leftarrow \text{COMPETE}$ ,  $R \leftarrow \emptyset$ , and  $W \leftarrow \emptyset$ .
18:   Choose priority  $p \in \{0, \dots, K-1\}$  uniformly at random.
19:   for all  $\ell \in L$  do SEND( $\text{request-lock}(p)$ ,  $\ell$ ).
20: CLEANUP:  $(\text{phase} \neq \perp) \vee (\text{state} = \text{UNLOCK}) \rightarrow$ 
21:   CLEANUP().
22: function CLEANUP() ▷ Helper function for processing disconnections  $D$ .
23:   for all  $\ell \in D$  do
24:     if  $\text{lock} = \ell$  then  $\text{lock} \leftarrow \perp$ .
25:     Remove  $\ell$  from all sets:  $L \leftarrow L \setminus \{\ell\}$ ,  $R \leftarrow R \setminus \{\ell\}$ ,  $W \leftarrow W \setminus \{(\ell, \cdot)\}$ ,  $H \leftarrow H \setminus \{\ell\}$ ,
26:      $A \leftarrow A \setminus \{\ell\}$ ,  $C \leftarrow C \setminus \{\ell\}$ , and  $P \leftarrow P \setminus \{(\ell, \cdot)\}$ .
27:   if  $C = \emptyset$  then
28:     for all  $\ell \in H$  do SEND( $\text{ready}()$ ,  $\ell$ ).
29:     Set  $A \leftarrow A \cup H$  and  $H \leftarrow \emptyset$ . ▷ All nodes on hold become applicants.
30:     if  $A \neq \emptyset$  then set  $\text{phase} \leftarrow \text{PREPARE}$ .
31:     else set  $\text{phase} \leftarrow \perp$ .

```

least one $\text{win}(\text{FALSE})$ message, it lost this competition and must compete again. Otherwise, if all responses are $\text{win}(\text{TRUE})$, it advances to the WIN state and sends $\text{set-lock}()$ messages to all nodes of $L(u)$. Once it has received $\text{ack-lock}()$ responses from all nodes of $L(u)$, it advances to the LOCKED state indicating $L(u)$ now represents the lock set $\mathcal{L}(u)$.

A participant v is responsible for coordinating the competition among all initiators that want to lock v . To delineate successive competitions, v distinguishes among initiators that are *candidates* in the current competition, *applicants* that may join the current competition, and those that are *on hold* for the next competition. When v receives a $\text{prepare}()$ message from an initiator u , it either puts u on hold if a competition is already underway or adds u as an applicant and replies $\text{ready}()$ otherwise. Participant v promotes its applicants to candidates when v receives their $\text{request-lock}(p)$ messages. Once all such messages are received from the competition's candidates, v notifies the one with the unique highest priority of its success and all others of their failure (or, in the case of a tie, all candidates fail). A winning competitor is removed from the candidate set while all others remain to try again; once the candidate set is empty, v promotes all initiators that were on hold to applicants. Finally, when v receives a $\text{set-lock}()$ message, it sets its lock variable accordingly and acknowledges this with an $\text{ack-lock}()$ response.

Algorithm 2 The LOCK Operation: Competition Phase for Node u .

```

1: RECEiverequest: On receiving request-lock( $p$ ) via port  $\ell \rightarrow$ 
2:   CLEANUP().
3:   if  $\ell \in A$  then set  $A \leftarrow A \setminus \{\ell\}$  and  $C \leftarrow C \cup \{\ell\}$ .
4:   Set  $P \leftarrow P \cup \{(\ell, p)\}$  and phase  $\leftarrow$  COMPETE. ▷ Close competition.
5: CHECKPRIORITIES: (phase = COMPETE)  $\wedge$  ( $|C| = |P|$ )  $\rightarrow$  ▷ All priorities received.
6:   CLEANUP().
7:   if lock =  $\perp$  and  $\exists(\ell, p) \in P$  with a unique highest  $p$  then
8:     SEND(win(TRUE),  $\ell$ ) and SEND(win(FALSE),  $\ell'$ ) for all  $\ell' \in C \setminus \{\ell\}$ .
9:   else SEND(win(FALSE),  $\ell$ ) for all  $\ell \in C$ .
10:  Reset  $P \leftarrow \emptyset$ . ▷ Competition is over.
11: RECEivewin: On receiving win( $b$ ) via port  $\ell \rightarrow$ 
12:   CLEANUP().
13:   Set  $W \leftarrow W \cup \{(\ell, b)\}$ .
14: CHECKWIN: (state = COMPETE)  $\wedge$  ( $|W| = |L|$ )  $\rightarrow$  ▷ All win( $b$ ) replies received.
15:   CLEANUP().
16:   if  $\exists(\cdot, \text{FALSE}) \in W$  then ▷ Start new locking attempt.
17:     Choose priority  $p \in \{0, \dots, K-1\}$  uniformly at random.
18:     for all  $\ell \in L$  do SEND(request-lock( $p$ ),  $\ell$ ).
19:   else ▷ Succeeded in locking.
20:     Set state  $\leftarrow$  WIN and reset  $R \leftarrow \emptyset$ .
21:     for all  $\ell \in L$  do SEND(set-lock( $\ell$ ),  $\ell$ ).
22:   Reset  $W \leftarrow \emptyset$ .
23: RECEivewinlock: On receiving set-lock( $\ell$ ) via port  $\ell \rightarrow$ 
24:   Set lock  $\leftarrow \ell$  and  $C \leftarrow C \setminus \{\ell\}$ .
25:   CLEANUP().
26:   SEND(ack-lock( $\ell$ ),  $\ell$ ).
27: RECEivewinlockack: On receiving ack-lock( $\ell$ ) via port  $\ell \rightarrow$ 
28:   CLEANUP().
29:   Set  $R \leftarrow R \cup \{\ell\}$ .
30: CHECKDONE: (state = WIN)  $\wedge$  ( $R = L$ )  $\rightarrow$  ▷ All lock acknowledgements received.
31:   CLEANUP().
32:   Set state  $\leftarrow$  LOCKED and reset  $R = \emptyset$ .
33:   return  $L$ . ▷ Locking complete.

```

4 Analysis

In this section, we prove the following theorem.

► **Theorem 1.** *If all nodes start with the initial values given by Table 1, the algorithm satisfies the mutual exclusion and lockout freedom properties under semi-synchronous concurrency, requires $\mathcal{O}(\Delta)$ memory per node and messages of size $\Theta(1)$, and has at most two messages in transit along any edge at any time.*

The algorithm in Section 3 is written with respect to local port labels; for ease of presentation, we use the corresponding nodes throughout this analysis and write $X_i(u)$ to denote the local variable X of node u at the start of round i . We begin with two straightforward lemmas demonstrating the eventual execution of enabled actions.

► **Lemma 2.** *Apart from CLEANUP, every enabled action will eventually be executed.*

Algorithm 3 The UNLOCK Operation for Node u .

```

1: INITUNLOCK: On UNLOCK being called  $\rightarrow$  ▷ Initiator initiates an unlock.
2:   if  $\mathbf{state} = \mathbf{LOCKED}$  then ▷ Only one UNLOCK per successful LOCK.
3:     CLEANUP().
4:     Set  $\mathbf{state} \leftarrow \mathbf{UNLOCK}$  and reset  $R \leftarrow \emptyset$ .
5:     for all  $\ell \in L$  do SEND(release-lock(),  $\ell$ ).
6: RECEIVERELEASE: On receiving release-lock() via port  $\ell \rightarrow$ 
7:   CLEANUP().
8:   Set  $\mathbf{lock} \leftarrow \perp$  and SEND(ack-unlock(),  $\ell$ ).
9: RECEIVEACKUNLOCK: On receiving ack-unlock() via port  $\ell \rightarrow$ 
10:  CLEANUP().
11:  Set  $R \leftarrow R \cup \{\ell\}$ .
12: CHECKUNLOCKED: ( $\mathbf{state} = \mathbf{UNLOCK}$ )  $\wedge$  ( $R = L$ )  $\rightarrow$  ▷ All unlock acknowledgements received.
13:  CLEANUP().
14:  Reset  $\mathbf{state} \leftarrow \perp$  and  $R = \emptyset$ . ▷ Unlocking complete.

```

Proof. Any enabled RECEIVE* action whose guard depends only on the receipt of some message must eventually be executed because it is assumed that every message in transit is eventually processed (unless the edge is disconnected, at which point the message is lost and the action is no longer enabled). Thus, it remains to consider the CHECK* actions.

Suppose CHECKSTART is enabled for a node u in some round i ; i.e., $\mathbf{state}_i(u) = \mathbf{PREPARE}$ and $R_i(u) = L_i(u)$. When $\mathbf{state} = \mathbf{PREPARE}$, only CHECKSTART can change the \mathbf{state} variable or reset R to \emptyset . Any execution of the CLEANUP action does not change the \mathbf{state} variable and maintains $R(u) = L(u)$ since it removes any disconnected neighbors from both sets. So CHECKSTART remains continuously enabled and thus must eventually be executed by the weakly fair adversary. An analogous argument also applies to CHECKPRIORITIES, CHECKWIN, CHECKDONE, and CHECKUNLOCKED. ◀

Lemma 2 shows that an enabled action will eventually be executed, but we also need to know that the actions become enabled in the first place. One potential obstacle is that CHECK* actions by a node u need to receive all responses from the nodes in $L(u)$ before becoming enabled. If some of nodes in $L(u)$ disconnect and their corresponding response messages are lost, the CHECK* action may be disabled indefinitely. This is one role of the CLEANUP action: removing disconnections from the algorithm's variables so other actions stop waiting for neighbors that no longer exist. We call such an action *pre-enabled* if it is currently disabled but would become enabled after CLEANUP is executed.

► **Lemma 3.** *Every pre-enabled action eventually becomes enabled.*

Proof. Suppose that CHECKSTART is pre-enabled for node u . Then $\mathbf{state}(u) = \mathbf{PREPARE}$ and u must have sent a `prepare()` message to itself in its execution of INITLOCK. So RECEIVEPREPARE is enabled for u , and by Lemma 2 it is eventually executed, updating $\mathbf{phase}(u) = \mathbf{PREPARE}$. This enables CLEANUP for u , and it will remain enabled until executed because only CLEANUP itself can reset \mathbf{phase} to \perp . Thus, CLEANUP must eventually be executed by the weakly fair adversary, enabling the pre-enabled CHECKSTART.

An analogous argument also applies to CHECKPRIORITIES, CHECKWIN, and CHECKDONE. For CHECKUNLOCKED, the condition $\mathbf{state} = \mathbf{UNLOCK}$ in the guard of CLEANUP ensures that CHECKUNLOCKED is eventually enabled. ◀

We continue our investigation of possible deadlocks resulting from actions remaining disabled by considering concurrent competitions. An initiator node u is *competing* if and only if $\text{state}(u) = \text{COMPETE}$, i.e., if u has executed `CHECKSTART` but has not yet received all `win()` messages needed to execute `CHECKWIN`. We model dependencies between competing initiators and participants at the start of round i as a directed bipartite graph $\mathcal{D}_i = (\mathcal{I}_i \cup \mathcal{P}_i, E_i)$ where $\mathcal{I}_i = \{u : \text{state}_i(u) = \text{COMPETE}\}$ is the set of competing initiators and $\mathcal{P}_i = \{u : \exists v \in \mathcal{I}_i \text{ s.t. } u \in L_i(v)\}$ is the set of participants. We note that some nodes belong to both partitions and consider their initiator and participant versions distinct. For nodes $u \in \mathcal{I}_i$ and $v \in \mathcal{P}_i \cap L_i(u)$ for which $u = v$ or $(u, v) \in G_i$ (i.e., the edge exists in round i), the directed edge $(u, v) \in E_i$ if and only if u has not yet sent a `request-lock()` message to v in response to the latest `win()` message from v ; analogously, $(v, u) \in E_i$ if and only if v has not yet sent a `win()` message to u in response to the latest `request-lock()` message from u .

► **Lemma 4.** *For all rounds i , \mathcal{D}_i is acyclic.*

Proof. Initially, no node has yet called `LOCK` and thus \mathcal{D}_0 is empty and trivially acyclic. So suppose that \mathcal{D}_j remains acyclic for all rounds $0 \leq j \leq i - 1$ and consider the following events that may occur in round $i - 1$ to form \mathcal{D}_i .

- A node u executes `CHECKSTART`. Then (v, u) is added to \mathcal{D}_i for each $v \in L_{i-1}(u)$ that u sends `request-lock()` messages to. But u is a sink, so \mathcal{D}_i remains acyclic.
- A node u executes `CHECKWIN`. If there exists $(\cdot, \text{FALSE}) \in W_{i-1}(u)$, then (u, v) is removed from \mathcal{D}_i and (v, u) is added to \mathcal{D}_i for each $v \in L_{i-1}(u)$ that u once again sends `request-lock()` messages to. As in the first case, this makes u a sink and \mathcal{D}_i remains acyclic. Otherwise, if all $(\cdot, b) \in W_{i-1}(u)$ have $b = \text{TRUE}$, u has won its competition and sets $\text{state}_i(u) = \text{WIN}$, meaning $u \notin \mathcal{D}_i$. So \mathcal{D}_i remains acyclic in this case as well.
- A node u executes `CHECKPRIORITIES`. Then (u, v) is removed from \mathcal{D}_i and (v, u) is added to \mathcal{D}_i for each $v \in C_{i-1}(u)$ that u sends `win()` messages to. For \mathcal{D}_i to be acyclic, it suffices to show it does not contain any outgoing edges from u ; i.e., there are no nodes w such that $u \in L_i(w)$, w has sent u a `request-lock()` message, but u has not yet sent a `win()` response to w . Such a node w could only have sent u a `request-lock()` message if it had previously received a `ready()` message from u , which in turn could only have been sent by u if u had included w as an applicant in $A(u)$. Thus, on receipt of the first `request-lock()` message from w , u would have promoted w to a candidate in $C(u)$, which is precisely the set that u responds to when executing `CHECKPRIORITIES`. So u has no outgoing edges in \mathcal{D}_i , as desired.
- An edge $\{u, v\}$ is disconnected in the TVG \mathcal{G} , for $u \in \mathcal{I}_{i-1}$ and $v \in \mathcal{P}_{i-1}$. This disconnection is processed by the `CLEANUP` helper function, removing v from $L(u)$ and thus any (u, v) edge from \mathcal{D}_i during the next execution of `CHECKWIN` by u ; an analogous statement holds for edges (v, u) in the next execution of `CHECKPRIORITIES` by v . As the removal of an edge cannot create a cycle, \mathcal{D}_i remains acyclic.

Therefore, \mathcal{D}_i remains acyclic in all cases, as claimed. ◀

► **Lemma 5.** *Every competing initiator eventually receives a `win()` response from its participants; likewise, every participant eventually receives a `request-lock()` response from its competing initiator(s).*

Proof. Suppose to the contrary that there exists a competing initiator u that waits indefinitely for a `win()` response from some participant v . Then the edge $\{u, v\}$ must never be disconnected in the TVG \mathcal{G} and the directed edge (v, u) must remain indefinitely in \mathcal{D} . By Lemmas 2 and 3, v can only be prohibited from sending the requisite `win()` message

if CHECKPRIORITIES remains disabled for v indefinitely. This, in turn, is only possible if v waits indefinitely for a `request-lock()` response from some competing initiator $w \neq u$. This implies that $\{v, w\}$ is never disconnected in \mathcal{G} and the directed edge (w, v) remains indefinitely in \mathcal{D} . As before, Lemmas 2 and 3 can be applied iteratively to show that each node must be waiting on another. But since the set of nodes V is finite, some node must eventually be revisited, establishing a directed cycle in \mathcal{D} and contradicting Lemma 4. ◀

Lemma 5 directly implies the following corollary.

► **Corollary 6.** *Every competition trial of a competing initiator eventually completes.*

To demonstrate that our algorithm satisfies lockout freedom, it remains to show that every competing initiator u eventually wins a competition trial by receiving all `win(TRUE)` responses from $L(u)$. We first address the situation in which a competition trial of u is *open*, meaning none of the nodes $v \in L(u)$ are locked during the trial.

► **Lemma 7.** *If $K = \Theta(1)$, then an initiator that competes in an open competition trial infinitely often will eventually win a competition, with probability 1.*

Proof. Consider any competing initiator u and any open competition trial of u . By the start of its second competition trial, $u \in C(v)$ for all $v \in L(u)$, implying that `phase(v) = COMPETE` and no other nodes will be added to $C(v) \cup A(v)$ while u is still competing for v . Since $|L(u) \setminus \{u\}| \leq \Delta$ and $|C(v) \cup A(v) \setminus \{u\}| \leq \Delta$ for each $v \in L(u) \setminus \{u\}$, node u can be competing against $c \leq \Delta^2$ other nodes. Every node chooses its priority uniformly at random from $\{0, \dots, K-1\}$, so it follows from symmetry that the probability u has the highest priority in a given trial is at least $1/\Delta^2$. In general,

$$\begin{aligned} \frac{\Pr [p(u) \text{ highest} \mid p(u) \text{ unique}]}{\Pr [p(u) \text{ highest}]} &= \frac{\sum_{p=0}^{K-1} \Pr [p(u) = p \wedge \forall v \neq u : p(v) \leq p(u) \mid p(u) \text{ unique}]}{\sum_{p=0}^{K-1} \Pr [p(u) = p \wedge \forall v \neq u : p(v) \leq p(u)]} \\ &= \frac{\sum_{p=0}^{K-1} \frac{1}{K} \left(\frac{p}{K-1}\right)^c}{\sum_{p=0}^{K-1} \frac{1}{K} \left(\frac{p+1}{K}\right)^c} \geq \frac{\sum_{p=0}^{K-1} p^c}{\sum_{p=1}^K p^c} \geq \frac{(K-1)^c}{2K^c} \geq \frac{(1-1/K)^{\Delta^2}}{2} \end{aligned}$$

Furthermore, the probability that u has a unique priority is $(1-1/K)^c \geq (1-1/K)^{\Delta^2}$. Thus, the probability that u has the unique highest priority in a given open trial is

$$\begin{aligned} \Pr [p(u) \text{ highest} \wedge p(u) \text{ unique}] &= \Pr [p(u) \text{ highest} \mid p(u) \text{ unique}] \cdot \Pr [p(u) \text{ unique}] \\ &\geq \frac{(1-1/K)^{\Delta^2}}{2\Delta^2} \cdot (1-1/K)^{\Delta^2} = \frac{(1-1/K)^{2\Delta^2}}{2\Delta^2} > 0. \end{aligned}$$

Since this probability is strictly positive, the probability that u never has the unique highest priority in an infinite sequence of open competition trials is

$$\lim_{n \rightarrow \infty} (1 - \Pr [p(u) \text{ highest} \wedge p(u) \text{ unique}])^n \leq \lim_{n \rightarrow \infty} \left(1 - \frac{(1-1/K)^{2\Delta^2}}{2\Delta^2}\right)^n = 0.$$

Therefore, with probability 1 there must eventually be an open competition trial in which u has the unique highest priority. Because the trial is open, all $v \in L(u)$ have `lock(v) = ⊥` and thus will send `win(TRUE)` responses to u .² ◀

² This proof can be easily extended to show that if $K > \Delta^2$, u will win a competition within $\mathcal{O}(\Delta^2)$ open competition trials, in expectation. We chose to avoid this increase in message size requirements from $\Theta(1)$ to $\mathcal{O}(\log \Delta)$ since time complexity is not a focus of this work.

12:12 Local Mutual Exclusion for Dynamic, Limited Message Passing Systems

We next show that a competing initiator competes in an open trial infinitely often. Recall from Section 2.2 that a LOCK operation by node u succeeds once u obtains locks for its persistent neighborhood, and once obtained, these locks are eventually released via UNLOCK.

► **Lemma 8.** *Every competing initiator eventually wins a competition trial with probability 1.*

Proof. Suppose to the contrary that a competing initiator u competes in an infinite number of competition trials. Only a finite number of these trials can be open, since u would eventually win one of an infinite number of open trials with probability 1 by Lemma 7. So an infinite number of trials of u must be closed; i.e., there are an infinite number of trials in which at least one $v \in L(u)$ has $\text{lock}(v) \neq \perp$. Since $|L(u) \setminus \{u\}| \leq \Delta$, there must be a node $v \in L(u)$ that is locked infinitely often. But by the start of its second competition trial, $u \in C(v)$ and no other nodes will be added to $C(v) \cup A(v)$ while u is still competing for v . Thus, only the nodes in $C(v) \cup A(v)$ and the node that had already locked v when u was added to $C(v)$ could possibly lock v . But whenever v sets its locks in RECEIVESETLOCK, it removes the locking node from $C(v)$. Moreover, any node that obtains locks must eventually release them, by supposition. So the set of nodes that could lock v is monotonically decreasing and thus nodes in $C(v) \cup A(v) \setminus \{u\}$ cannot lock v an infinite number of times, a contradiction. ◀

For an initiator u to benefit from eventual victory ensured by Lemma 8, it must become competing in the first place; i.e., it must advance to $\text{state}(u) = \text{COMPETE}$.

► **Lemma 9.** *Every initiator eventually becomes competing.*

Proof. Suppose to the contrary that an initiator u never becomes competing, i.e., it never executes CHECKSTART. By Lemmas 2 and 3, this is only possible if CHECKSTART remains disabled indefinitely. To be an initiator at all, u must have executed INITLOCK, set $\text{state}(u) = \text{PREPARE}$, and sent `prepare()` messages to all nodes $v \in L(u)$. So u must be waiting for a `ready()` response from at least one $v \in L(u)$ that remains connected to u indefinitely.

By Lemma 2, such a node v must eventually execute RECEIVEPREPARE. During this execution, it must be the case that $\text{phase}(v) = \text{COMPETE}$ and v adds u to $H(v)$; otherwise, v would have added u to $A(v)$ and replied to u with a `ready()` message, a contradiction. Only the CLEANUP helper function can reset $\text{phase}(v)$ to \perp , but it only does so when $C(v) \cup A(v) \cup H(v) = \emptyset$ which is not the case since $u \in H(v)$. So the CLEANUP action is continuously enabled for v and is eventually executed by the weakly fair adversary. During this execution, it must be the case that $C(v) \neq \emptyset$; otherwise, v would have sent `ready()` messages to all initiators on hold at v , including u , a contradiction. But for this situation to occur indefinitely, there must exist some competitor in the finite set $C(v) \cup A(v)$ that competes in an infinite number of trials, a contradiction of Lemma 8. ◀

Combining Corollary 6 with Lemmas 8 and 9 implies the following corollary.

► **Corollary 10.** *The local mutual exclusion algorithm satisfies lockout freedom.*

Recall from Section 2.2 that the mutual exclusion property is trivially satisfied by our construction of the lock sets. Thus, we conclude the proof of Theorem 1 with the following result regarding the algorithm's memory and message size requirements.

► **Lemma 11.** *The algorithm requires $\mathcal{O}(\Delta)$ memory per node and messages of size $\Theta(1)$, and there are at most two messages in transit along any given edge at any time.*

Proof. Table 1 shows that `phase` and `state` can be stored in $\Theta(1)$ bits each and `lock` can be stored in $\log_2 \Delta$ bits. The remaining variables can be represented as linear registers of length Δ , where port ℓ is in the set variable X if and only if the ℓ -th bit of register X is `TRUE`. So the memory bound of $\mathcal{O}(\Delta)$ follows. Similarly, there are a constant number of message types, among which only `request-lock()` and `win()` carry additional data. A `win()` message carries one bit signaling whether a competition trial was won or lost. A `request-lock()` message carries a randomly chosen priority, which by Lemma 7 can be stored in $\Theta(1)$ bits.

To bound the number of messages in transit per edge per time, consider the execution of a `LOCK` operation by an initiator node u . The local mutual exclusion algorithm is structured around pairs of initiator messages and participant responses: `prepare()/ready()` messages in the preparation phase, `request-lock()/win()` messages in each competition trial, and `set-lock()/ack-lock()` messages once a node has won a trial. In each scenario, only one message per pair is in transit along $\{u, v\}$ per time for each $v \in L(u)$. Moreover, node u does not advance to the next phase and send any additional messages until all messages of the current phase are processed. An analogous argument applies to the `UNLOCK` operation with its `release-lock()/ack-unlock()` message pairs. Thus, there can be at most one message in transit per edge per time involved with any initiator's `LOCK` or `UNLOCK` operation.

Furthermore, an initiator u can execute at most one `LOCK` or `UNLOCK` operation per time since u can only start a `LOCK` operation by executing `INITLOCK` if `state(u) = \perp` , implying it holds no locks; similarly, u can only start an `UNLOCK` operation by executing `INITUNLOCK` if `state(u) = LOCKED`, implying its previous `LOCK` operation has succeeded.

Thus, the lemma follows since there are at most two initiators u and v per edge $\{u, v\}$. ◀

5 Extending to Asynchronous Concurrency

Section 4 proved Theorem 1 under semi-synchronous concurrency in which (i) topological changes occur at discrete times in between rounds of action executions and (ii) the adversary chooses any non-empty subset of nodes to act in each round and those nodes' action executions are guaranteed to end before the next round begins. In this section, we prove that Theorem 1 holds even in the more general asynchronous setting.

All assumptions from Section 2.1 about the time-varying graph \mathcal{G} , the nodes, their asynchronous message passing, and the structure of algorithms and their actions remain the same. However, in an *asynchronous schedule*, the adversary can schedule action executions over arbitrary finite time intervals, including those that are concurrent with topological changes and span multiple TVG rounds. In this setting, our prior assumptions about the disconnection detector now imply that any topological changes incident to u that are concurrent with one of its action executions are not observed or processed by u until its next action execution. We further assume for the asynchronous setting that any message sent by node u during one of its action executions starting at time t_1 is processed by a node v during some other action execution starting at time $t_2 > t_1$ if and only if the edge $\{u, v\} \in G_t$ for all $t \in [t_1, t_2]$. This implies that when an edge is disconnected, all messages in transit along that edge are immediately lost and no further messages can be sent or received by the corresponding ports until the corresponding action executions have finished.

► **Lemma 12.** *For any asynchronous schedule \mathcal{S} , there exists a semi-synchronous schedule \mathcal{S}' containing the same action executions as in \mathcal{S} that produces the same outcome for every action execution in \mathcal{S} .*

Proof. Consider any asynchronous schedule \mathcal{S} of the local mutual exclusion algorithm and let \mathcal{E} be the set of all action executions in \mathcal{S} . Analogous to Lamport [34], we define the *causal relation* \rightarrow on \mathcal{E} as the smallest relation satisfying the following three conditions:

- If $\alpha \in \mathcal{E}$ is an execution by node u and $\beta \in \mathcal{E}$ is the next execution by u , then $\alpha \rightarrow \beta$.
- If a message sent in $\alpha \in \mathcal{E}$ is processed in $\beta \in \mathcal{E}$, then $\alpha \rightarrow \beta$.
- If $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$.

Since all causal relations are naturally forward in time, it follows that the graph represented by the causal relations on \mathcal{E} forms a DAG. Thus, the action executions of \mathcal{E} can be topologically sorted in some order $[\alpha_1, \alpha_2, \dots]$.

Now, consider the schedule $\hat{\mathcal{S}}$ containing the same action executions starting at the same times as those in \mathcal{S} , but (i) each action execution takes 0 time and (ii) any set of action executions starting at the same time as some edge changes is shifted before these edge changes without changing the order of the action executions. Then $\hat{\mathcal{S}}$ can be transformed into a semi-synchronous schedule \mathcal{S}' by adding filler time steps when no edges change so that each node executes at most one action per round and all action executions between two time steps start at the same time. Certainly, \mathcal{S}' is still a valid schedule since all causal relations remain forward in time and – by our assumption on asynchronous message processing – any message sent by action execution α that is processed by action execution β in \mathcal{S} can still be processed by β in \mathcal{S}' . Furthermore, since the causal relations haven't changed, the action executions in \mathcal{S}' can be sorted in the same order $[\alpha_1, \alpha_2, \dots]$ as for \mathcal{S} . Since any action execution can only change a node's state or send messages and, in both schedules, it only sees a snapshot of D at its start, it follows by induction on the ordering of the action executions that for any i , the outcome of α_i is identical in \mathcal{S} and \mathcal{S}' . ◀

As in the semi-synchronous setting, the mutual exclusion property is trivially satisfied in the asynchronous setting. But suppose to the contrary that there exists an asynchronous schedule in which at least one LOCK operation never succeeds. Lemma 12 shows that there must exist a semi-synchronous schedule in which at least one LOCK operation never succeeds, contradicting Theorem 1. So we have the following corollary.

► **Corollary 13.** *The local mutual exclusion algorithm also satisfies the mutual exclusion and lockout freedom properties under any asynchronous schedule.*

6 Applications

We next establish how our algorithm for local mutual exclusion can be used to implement key assumptions present in formal models of dynamic distributed systems. In particular, we focus on the assumptions of independent pairwise interactions in *population protocols* [3] and concurrency control operations in the *canonical amoebot model* of programmable matter [17].

Population Protocols. Inspired by passively mobile sensor networks, Angluin et al. [3] proposed the *population protocols* model. Each agent in a population is assumed to have a finite state and a transition function defining how that state evolves as a result of a pairwise interaction with another agent. Agents cannot explicitly control their movements or who they interact with; i.e., they are passively dynamic. Instead, it is typically assumed that a sequential scheduler chooses one pair of agents to interact per time step. In reality, however, many agents within interacting distance might exist concurrently (see, e.g., [16]), requiring a mechanism to organize these agents into a matching of independent pairs.

This goal could be achieved directly using our algorithm for local mutual exclusion. Any agent u that wants to interact must first call LOCK. On success, u then chooses any locked neighbor to interact with, if it has one; if desired, one could even generalize the usual pairwise interactions to interactions among the full group of locked neighbors. Lockout freedom ensures u will eventually be allowed to make this choice, and mutual exclusion ensures this pairwise interaction is isolated from any others. After interacting, u then releases its locks with UNLOCK. If the expected number of competing agents is high, an alternative implementation of our algorithm could have u make its choice of interacting neighbor v first and then try to lock only u and v to avoid a lengthy competition. On success, u would then interact with v , isolated from any other interactions, and then unlock itself and v . In both implementations, it is possible that all neighbors may move out of interaction range, leaving u to lock only itself. In this situation, no interaction occurs and u simply unlocks itself.

Both implementations require $\mathcal{O}(\Delta)$ memory per agent and messages of size $\Theta(1)$. For many applications of population protocols where Δ is a fixed constant (e.g., proximity graphs or IoT), these requirements are reduced to $\Theta(1)$. Thus, our algorithm for local mutual exclusion could provide isolated pairwise interactions assumed by population protocols even in the presence of underlying network dynamics and asynchronous concurrency.

The Canonical Amoebot Model. The *amoebot model* abstracts active programmable matter as a collection of simple computational elements called *amoebots* that move and interact locally to collectively achieve tasks of coordination and movement. Each amoebot is typically assumed to be anonymous and have only constant-size memory, but can control its movements. The *canonical amoebot model* [17] is an updated formalization that addresses concurrency by partitioning amoebot functionality into a high-level application layer where algorithms call various operations and a low-level system layer where those operations are executed via asynchronous message passing. Two such operations are the concurrency control operations, LOCK and UNLOCK, which are used in a *concurrency control framework* to convert amoebot algorithms that terminate in the sequential setting and satisfy certain conventions into algorithms that exhibit equivalent behavior in the concurrent setting [17].

The canonical amoebot model treats the LOCK and UNLOCK operations as black boxes without giving an implementation. These operations facilitate amoebots gaining exclusive access to themselves and their neighbors, much like our mutual exclusion property, and are assumed to terminate (either successfully or in failure) in finite time. The asynchronous extension of our local mutual exclusion algorithm presented in Section 5 could directly implement these operations, ensuring isolation of concurrent amoebot actions even as connections between amoebots change due to their movements. One interesting feature of such an implementation is that while the amoebot LOCK operation is allowed to fail – which must be taken into account by algorithm designers – our LOCK operation always succeeds due to lockout freedom, reducing complexity in algorithm design. Moreover, for the often-considered geometric space variant in which an (expanded) amoebot can have at most eight neighbors, our algorithm has $\Theta(1)$ amoebot memory and message size requirements.

7 Conclusion

We presented an algorithm for local mutual exclusion that enables weakly capable nodes to isolate concurrent actions involving their persistent neighborhoods despite dynamic network topology. Our algorithm ensures that nodes belong to at most one locked neighborhood at a time (mutual exclusion) and that every lock request eventually succeeds (lockout freedom).

It requires $\mathcal{O}(\Delta)$ memory per node and messages of size $\Theta(1)$ – where Δ is the maximum number of connections per node – and is compatible with anonymous, message passing nodes that operate semi-synchronously or asynchronously. These weak requirements make our algorithm suitable for a wide range of application domains such as overlay networks, IoT, modular robots, and programmable matter. As two concrete examples, we demonstrated how our algorithm could implement the pairwise interactions assumed by population protocols [3] and the concurrency control operations assumed by the canonical amoebot model [17].

References

- 1 Karine Altisen, Stéphane Devismes, Anaïs Durand, Colette Johnen, and Franck Petit. Self-stabilizing Systems in Spite of High Dynamics. In *International Conference on Distributed Computing and Networking 2021*, pages 156–165, 2021. doi:10.1145/3427796.3427838.
- 2 Marta Andrés Arroyo, Sarah Cannon, Joshua J. Daymude, Dana Randall, and Andréa W. Richa. A stochastic approach to shortcut bridging in programmable matter. *Natural Computing*, 17(4):723–741, 2018. doi:10.1007/s11047-018-9714-x.
- 3 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006. doi:10.1007/s00446-005-0138-3.
- 4 Hagit Attiya, Alex Kogan, and Jennifer L. Welch. Efficient and Robust Local Mutual Exclusion in Mobile Ad Hoc Networks. *IEEE Transactions on Mobile Computing*, 9(3):361–375, 2010. doi:10.1109/TMC.2009.137.
- 5 Baruch Awerbuch, Michael Luby, Andrew V. Goldberg, and Serge A. Plotkin. Network decomposition and locality in distributed computation. In *30th Annual Symposium on Foundations of Computer Science*, pages 364–369, 1989. doi:10.1109/SFCS.1989.63504.
- 6 Roberto Baldoni, Antonino Virgillito, and Roberto Petrassi. A distributed mutual exclusion algorithm for mobile ad-hoc networks. In *Proceedings ISCC 2002 Seventh International Symposium on Computers and Communications*, pages 539–544, 2002. doi:10.1109/ISCC.2002.1021727.
- 7 Mahfoud Benchaïba, Abdelmadjid Bouabdallah, Nadjib Badache, and Mohamed Ahmed-Nacer. Distributed Mutual Exclusion Algorithms in Mobile Ad Hoc Networks: An Overview. *ACM SIGOPS Operating Systems Review*, 38(1):74–89, 2004. doi:10.1145/974104.974111.
- 8 Michael A. Bender, Martin Farach-Colton, Simai He, Bradley C. Kuszmaul, and Charles E. Leiserson. Adversarial Contention Resolution for Simple Channels. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 325–332, 2005. doi:10.1145/1073970.1074023.
- 9 Federico Cali, Marco Conti, and Enrico Gregori. IEEE 802.11 Protocol: Design and Performance Evaluation of an Adaptive Backoff Mechanism. *IEEE Journal on Selected Areas in Communications*, 18(9):1774–1786, 2000. doi:10.1109/49.872963.
- 10 John I. Capetanakis. Tree Algorithms for Packet Broadcast Channels. *IEEE Transactions on Information Theory*, 25(5):505–515, 1979. doi:10.1109/TIT.1979.1056093.
- 11 Arnaud Casteigts. A Journey through Dynamic Networks (with Excursions), 2018. HDR, available online at <https://hal.archives-ouvertes.fr/tel-01883384/>.
- 12 Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-Varying Graphs and Dynamic Networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012. doi:10.1080/17445760.2012.668546.
- 13 Cameron Chalk, Austin Luchsinger, Eric Martinez, Robert Schweller, Andrew Winslow, and Tim Wylie. Freezing Simulates Non-freezing Tile Automata. In *DNA Computing and Molecular Programming*, volume 11145 of *Lecture Notes in Computer Science*, pages 155–172, 2018. doi:10.1007/978-3-030-00030-1_10.

- 14 Arjun Chandrasekhar, Deborah M. Gordon, and Saket Navlakha. A distributed algorithm to maintain and repair the trail networks of arboreal ants. *Scientific Reports*, 8(1):9297, 2018. doi:10.1038/s41598-018-27160-3.
- 15 Yu Chen and Jennifer L. Welch. Self-stabilizing dynamic mutual exclusion for mobile ad hoc networks. *Journal of Parallel and Distributed Computing*, 65(9):1072–1089, 2005. doi:10.1016/j.jpdc.2005.03.009.
- 16 Artur Czumaj and Andrzej Lingas. On Truly Parallel Time in Population Protocols. Available online at [arXiv:2108.11613](https://arxiv.org/abs/2108.11613), 2021.
- 17 Joshua J. Daymude, Andréa W. Richa, and Christian Scheideler. The Canonical Amoebot Model: Algorithms and Concurrency Control. In *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:19, 2021. doi:10.4230/LIPIcs.DISC.2021.20.
- 18 Michael J. Demmer and Maurice P. Herlihy. The arrow distributed directory protocol. In Shay Kutten, editor, *Distributed Computing*, volume 1499 of *Lecture Notes in Computer Science*, pages 119–133, 1998. doi:10.1007/BFb0056478.
- 19 Zahra Derakhshandeh, Shlomi Dolev, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Amoebot - a new model for programmable matter. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 220–222, 2014. doi:10.1145/2612669.2612712.
- 20 E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965. doi:10.1145/365559.365617.
- 21 Yuval Emek and Roger Wattenhofer. Stone age distributed computing. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, pages 137–146, Montréal, Québec, Canada, 2013. ACM. doi:10.1145/2484239.2484244.
- 22 Javier Esparza and Fabian Reiter. A Classification of Weak Asynchronous Models of Distributed Computing. In *31st International Conference on Concurrency Theory (CONCUR 2020)*, volume 171 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.CONCUR.2020.10.
- 23 Michael Feldmann, Christian Scheideler, and Stefan Schmid. Survey on Algorithms for Self-stabilizing Overlay Networks. *ACM Computing Surveys*, 53(4):74:1–74:24, 2020. doi:10.1145/3397190.
- 24 Michael J. Fischer, Nancy A. Lynch, James E. Burns, and Allan Borodin. Resource allocation with immunity to limited process failure. In *20th Annual Symposium on Foundations of Computer Science (SFCS 1979)*, pages 234–254, 1979. doi:10.1109/SFCS.1979.37.
- 25 Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, editors. *Distributed Computing by Mobile Entities: Current Research in Moving and Computing*, volume 11340 of *Lecture Notes in Computer Science*. Springer International Publishing, Cham, 2019. doi:10.1007/978-3-030-11072-7.
- 26 Mohsen Ghaffari, Cameron Musco, Tsvetomira Radeva, and Nancy Lynch. Distributed House-Hunting in Ant Colonies. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 57–66, 2015. doi:10.1145/2767386.2767426.
- 27 Abdolhamid Ghodselahi and Fabian Kuhn. Dynamic Analysis of the Arrow Distributed Directory Protocol in General Networks. In *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:16, 2017. doi:10.4230/LIPIcs.DISC.2017.22.
- 28 Heiko Hamann. *Swarm Robotics: A Formal Approach*. Springer International Publishing, Cham, 2018. doi:10.1007/978-3-319-74528-2.
- 29 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991. doi:10.1145/114005.102808.
- 30 Yuh-Jzer Joung. Asynchronous group mutual exclusion. *Distributed Computing*, 13(4):189–206, 2000. doi:10.1007/PL00008918.

- 31 Yuh-Jzer Joung. The congenial talking philosophers problem in computer networks. *Distributed Computing*, 15(3):155–175, 2002. doi:10.1007/s004460100069.
- 32 Pankaj Khanchandani and Roger Wattenhofer. The Arvy Distributed Directory Protocol. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 225–235, 2019. doi:10.1145/3323165.3323181.
- 33 Fabian Kuhn, Yannic Maus, and Simon Weidner. Deterministic Distributed Ruling Sets of Line Graphs. In *Structural Information and Communication Complexity*, volume 11085 of *Lecture Notes in Computer Science*, pages 193–208, 2018. doi:10.1007/978-3-030-01325-7_19.
- 34 Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978. doi:10.1145/359545.359563.
- 35 Mamoru Maekawa. A \sqrt{N} algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, 1985. doi:10.1145/214438.214445.
- 36 Othon Michail, George Skretas, and Paul G. Spirakis. Distributed Computation and Reconfiguration in Actively Dynamic Networks. *Distributed Computing*, 2021. doi:10.1007/s00446-021-00415-5.
- 37 Othon Michail and Paul G. Spirakis. Simple and efficient local codes for distributed stable network construction. *Distributed Computing*, 29(3):207–237, 2016. doi:10.1007/s00446-015-0257-4.
- 38 Othon Michail and Paul G. Spirakis. Connectivity preserving network transformers. *Theoretical Computer Science*, 671:36–55, 2017. doi:10.1016/j.tcs.2016.02.040.
- 39 Matthew J. Patitz. An introduction to tile-based self-assembly and a survey of recent results. *Natural Computing*, 13(2):195–224, 2014. doi:10.1007/s11047-013-9379-4.
- 40 Benoit Piranda and Julien Bourgeois. Designing a quasi-spherical module for a huge modular robot to create programmable matter. *Autonomous Robots*, 42:1619–1633, 2018. doi:10.1007/s10514-018-9710-0.
- 41 Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, 1989. doi:10.1145/58564.59295.
- 42 Michel Raynal and Gadi Taubenfeld. Mutual exclusion in fully anonymous shared memory systems. *Information Processing Letters*, 158:105938, 2020. doi:10.1016/j.ip1.2020.105938.
- 43 Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981. doi:10.1145/358527.358537.
- 44 Johannes Schneider, Michael Elkin, and Roger Wattenhofer. Symmetry breaking depending on the chromatic number or the neighborhood growth. *Theoretical Computer Science*, 509:40–50, 2013. doi:10.1016/j.tcs.2012.09.004.
- 45 Bharti Sharma, Ravinder Singh Bhatia, and Awadhesh Kumar Singh. A Token Based Protocol for Mutual Exclusion in Mobile Ad Hoc Networks. *Journal of Information Processing Systems*, 10(1):36–54, 2014. doi:10.3745/JIPS.2014.10.1.036.
- 46 Harisu Abdullahi Shehu, Md. Haidar Sharif, and Rabie A. Ramadan. Distributed Mutual Exclusion Algorithms for Intersection Traffic Problems. *IEEE Access*, 8:138277–138296, 2020. doi:10.1109/ACCESS.2020.3012573.
- 47 Mukesh Singhal. A Taxonomy of Distributed Mutual Exclusion. *Journal of Parallel and Distributed Computing*, 18(1):94–101, 1993. doi:10.1006/jpdc.1993.1048.
- 48 David Soloveichik, Matthew Cook, Erik Winfree, and Jehoshua Bruck. Computation with finite stochastic chemical reaction networks. *Natural Computing*, 7(4):615–633, 2008. doi:10.1007/s11047-008-9067-y.
- 49 Lili Su, Chia-Jung Chang, and Nancy Lynch. Spike-Based Winner-Take-All Computation: Fundamental Limits and Order-Optimal Circuits. *Neural Computation*, 31(12):2523–2561, 2019. doi:10.1162/neco_a_01242.
- 50 Jennifer E. Walter, Jennifer L. Welch, and Nitin H. Vaidya. A Mutual Exclusion Algorithm for Ad Hoc Mobile Networks. *Wireless Networks*, 7:585–600, 2001. doi:10.1023/A:1012363200403.

- 51 Damien Woods, Ho-Lin Chen, Scott Goodfriend, Nadine Dabby, Erik Winfree, and Peng Yin. Active self-assembly of algorithmic shapes and patterns in polylogarithmic time. In *Proceedings of the 4th Conference on Innovations in Theoretical Computer Science*, pages 353–354, 2013. doi:10.1145/2422436.2422476.
- 52 Weigang Wu, Jiebin Zhang, Aoxue Luo, and Jiannong Cao. Distributed Mutual Exclusion Algorithms for Intersection Traffic Control. *IEEE Transactions on Parallel and Distributed Systems*, 26(1):65–74, 2015. doi:10.1109/TPDS.2013.2297097.
- 53 Mark Yim, Wei-Min Shen, Behnam Salemi, Daniela Rus, Mark Moll, Hod Lipson, Eric Klavins, and Gregory S. Chirikjian. Modular Self-Reconfigurable Robot Systems [Grand Challenges of Robotics]. *IEEE Robotics & Automation Magazine*, 14(1):43–52, 2007. doi:10.1109/MRA.2007.339623.

Dynamic Size Counting in Population Protocols

David Doty   

University of California, Davis, CA, USA

Mahsa Eftekhari   

University of California, Davis, CA, USA

Abstract

The population protocol model describes a network of anonymous agents that interact asynchronously in pairs chosen at random. Each agent starts in the same initial state s . We introduce the *dynamic size counting* problem: approximately counting the number of agents in the presence of an adversary who at any time can remove any number of agents or add any number of new agents in state s . A valid solution requires that after each addition/removal event, resulting in population size n , with high probability each agent “quickly” computes the same constant-factor estimate of the value $\log_2 n$ (how quickly is called the *convergence time*), which remains the output of every agent for as long as possible (the *holding time*). Since the adversary can remove agents, the holding time is necessarily finite: even after the adversary stops altering the population, it is impossible to *stabilize* to an output that never again changes.

We first show that a protocol solves the dynamic size counting problem if and only if it solves the *loosely-stabilizing counting* problem: that of estimating $\log n$ in a *fixed-size* population, but where the adversary can initialize each agent in an arbitrary state, with the same convergence time and holding time. We then show a protocol solving the loosely-stabilizing counting problem with the following guarantees: if the population size is n , M is the largest initial estimate of $\log n$, and s is the maximum integer initially stored in any field of the agents’ memory, we have expected convergence time $O(\log n + \log M)$, expected polynomial holding time, and expected memory usage of $O(\log^2(s) + (\log \log n)^2)$ bits. Interpreted as a dynamic size counting protocol, when changing from population size n_{prev} to n_{next} , the convergence time is $O(\log n_{\text{next}} + \log \log n_{\text{prev}})$.

2012 ACM Subject Classification Theory of computation \rightarrow Distributed algorithms; Theory of computation \rightarrow Models of computation

Keywords and phrases Loosely-stabilizing, population protocols, size counting

Digital Object Identifier 10.4230/LIPIcs.SAND.2022.13

Related Version *Full Version*: <https://arxiv.org/abs/2202.12864>

Supplementary Material *Software (Simulation Results with Colab Notebook)*:

https://github.com/eftekhari-mhs/population-protocols/tree/main/dynamic_counting
archived at `swh:1:dir:a71288ec3836738d716285e3e6f6446978940c2f`

Funding Supported by NSF award 1900931 and CAREER award 1844976.

1 Introduction

A population protocol [6] is a network of n anonymous and identical *agents* with finite memory called the *state*. A scheduler repeatedly selects a pair of agents independently and uniformly at random to interact. Each agent sees the entire state of the other agent in the interaction and updates own state in response. Time complexity is measured by *parallel time*: the number of interactions divided by the population size n , capturing the natural time scale in which each agent has $\Theta(1)$ interactions per unit time. The agents collectively do a computation, e.g., population size counting: computing the value n . Counting is a fundamental task in distributed computing: knowing an estimate of n often simplifies the design of protocols solving problems such as majority and leader election [1, 2, 4, 11, 13–16, 24, 31, 35].



© David Doty and Mahsa Eftekhari;

licensed under Creative Commons License CC-BY 4.0

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).

Editors: James Aspnes and Othon Michail; Article No. 13; pp. 13:1–13:18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

13:2 Dynamic Size Counting in Population Protocols

A protocol is defined by a *transition function* with a pair of states as input and as output (more generally to capture randomized protocols, a relation that can associate multiple outputs to the same input). For example, consider the simple counting protocol with transitions $L_i, L_j \rightarrow L_{i+j}, F_{i+j}$, with every agent starting in L_1 . In population size n , this protocol converges to a single agent in state L_n , with all other agents in state F_i for some i . The additional transitions $F_i, F_j \rightarrow F_j, F_j$ for $i < j$ propagate the output n to all agents.

The dynamic size counting problem

In contrast to most work, which assumes the population size n is fixed over time, we model an adversary that can add or remove agents arbitrarily and repeatedly during the computation. All agents start in the same state, including newly added agents. The goal is for each agent to approximately count the population size n , which we define to mean that all agents should eventually store the same output k in their states, which with high probability is within a constant multiplicative factor of $\log n$.¹ Once all agents have the same output k , they have *converged*. They maintain k as the output for some time called the *holding time* (after which they might alter k even if the population size has not changed). In response to a “significant” change in size from n_{prev} to n_{next} , agents should re-converge to a new output k' of $\log n_{\text{next}}$. (Agents are not “notified” about the change; instead they must continually monitor the population to test whether their current output is accurate.) Note that if n_{prev} is close to n_{next} (within a polynomial factor), then k may remain an accurate estimate of n_{next} , so agents may not re-converge in response to a small change.

Ideally the expected convergence time is small, and the expected holding time is large. With a fixed size population, it is common to require the output to *stabilize* to a value that never again changes after convergence, i.e., infinite holding time. However, this turns out to be impossible with an adversary that can remove agents (Observation 3.4). When changing from size n_{prev} to n_{next} , our protocol achieves expected convergence time $O(\log n_{\text{next}} + \log \log n_{\text{prev}})$ and expected holding time $\Omega(n_{\text{next}}^c)$, where c can be made arbitrarily large. The number of bits of memory used per agent is $O(\log^2(s) + (\log \log n)^2)$, where s is the maximum integer stored in the agents’ memory after the change.

While it is common to measure population protocol memory complexity by counting the number of states (which is exponentially larger than the number of bits required to represent each state), that measure is a bit awkward here. Our protocol is uniform – the same transition rules for every population size – so has an infinite number of producible states. One could count expected number of states that will be produced, but this is a bit misleading: in time t each agent visits $O(t)$ states on average, so $O(t \cdot n)$ states total. Counting how many bits are required is more accurate metric of the actual memory requirements.

The loosely-stabilizing counting problem

The dynamic size counting problem has an equivalent characterization: rather than removing agents and adding them with a fixed initial state, the *loosely-stabilizing* adversary sets each agent to an arbitrary initial state in a fixed-size population. A protocol solves the dynamic size counting problem if and only if it solves the loosely-stabilizing counting problem, with the same convergence and holding times (Lemma 3.5). Due to this equivalence, we analyze

¹ *Nonuniform* protocols require agents to be initialized with an estimate k of $\log n$ in order to accomplish other tasks, such as a “leaderless phase clock” [1]. The bound $k = \Theta(\log n)$ is necessary and sufficient for correctness and speed in most cases [1, 2, 4, 11, 13–16, 24, 31, 35].

our protocol assuming a fixed population size and adversarial initial states. In this case our convergence time $O(\log n + \log M)$ is measured as a function of the population size n and the value M that is the maximum `estimate` value stored in agents' memory. From the perspective of the dynamic size counting problem, these "adversarial initial states" would correspond to the agent states after correctly estimating the *previous* population size, just prior to adding or removing agents.

1.1 Related work

Initialized counting with a fixed size population. In population protocols with fixed size, there is work computing exactly or approximately the population size n . For a full review see [19]. Such protocols reach a *stable* configuration from which the output cannot change. Some of these counting protocols would still solve the counting problem in the presence of an adversary who can only add agents (see Observation 3.3). However, these protocols fail in the presence of an adversary who can also remove agents, since they work only in the initialized setting and rely on reaching a stable configuration (see Observation 3.4).

Self-stabilizing counting with a fixed size population. A population protocol is *self-stabilizing* if, from *any* initial configuration, it reaches to a correct stable configuration. Self-stabilizing size counting has been studied [8–10,27], but provably requires adding a "base station" agent that cannot be corrupted by the adversary. In these protocols the base station is the only agent required to learn the population size. Aspnes, Beauquier, Burman, and Sohler [8] showed a time- and space-optimal protocol that solves the exact counting problem in $O(n \log n)$ time, using 1-bit memory for each non-base station agent.

Size regulation in a dynamically sized population. The model described by Goldwasser, Ostrovsky, Scafuro, and Sealton [25] is close to our setting. They consider the *size regulation problem*: approximately maintaining a target size (hard-coded into each agent) using $O(\log \log n)$ bits of memory per agent, despite an adversary that (like ours) adds or removes agents. That paper assumes a model variation in which:

- The agents can replicate or self-destruct.
- The computation happens through synchronized rounds of interactions. At each round the scheduler selects a random matching of size $k = O(n)$ agents to interact.
- The adversary's changes to the population size are limited. The adversary can insert or delete a total of $o(n^{1/4})$ agents within each round.

The latter two model differences above crucially rule out their protocol as useful for our problem. We use the standard asynchronous scheduler, and much of the complexity of our protocol is to handle drastic population size changes (e.g., removing $n - \log n$ agents). Additionally, their protocol heavily relies on flipping coins of bias $\frac{1}{\sqrt{n}}$ that we cannot utilize since the agents don't start with an estimate of n . Moreover, even when the agents compute their estimate, the population size might change.

Loosely-stabilizing leader election. Sudo, Nakamura, Yamauchi, Ooshita, Kakugawa, and Masuzawa [34] introduce loose-stabilization as a relaxation for the self-stabilizing leader election problem in which the agents must know the exact population size to elect a leader. The loosely stabilizing leader election guarantees that starting from any configuration, the population will elect a leader within a short time. After that, the agents hold the leader for a long time but not forever (in contrast with self-stabilization). On the positive side, the agents no longer need to know the exact population size to solve the loosely-stabilizing

leader election, but a rough upper bound suffices. Loosely-stabilizing leader election has been studied, providing a time-optimal protocol that solves the leader election problem [33] and a tradeoff between the holding and convergence times [26, 36].

Computation with dynamically changing inputs. Alistarh, Töpfer, and Uznański [5] consider the dynamic variant of the comparison problem. In the comparison problem, a subset of population are in the input states X and Y and the goal is to compute if $X > Y$ or $X < Y$. In the dynamic variant of the comparison problem, they assume an adversary who can change the counts of the input states at any time. The agents should compute the output as long as the counts remain untouched for sufficiently long time. They propose a protocol that solves the comparison problem in $O(\log n)$ time using $O(\log n)$ states per agent, assuming $|X| \geq C_2 \cdot |Y| \geq C_1 \log n$ for some constants $C_1, C_2 > 1$.

Berenbrink, Biermeier, Hahn, and Kaaser [12] consider the adaptive majority problem (generalization of the comparison problem [5]). At any time every agent has an opinion from $\{X, Y\}$ or undecided and their opinions might change adversarially. The goal is to have agreement in the population about the majority opinion. They introduce a non-uniform loosely-stabilizing leaderless phase clock that that uses $O(\log n)$ states to solve the adaptive majority problem. This is similar to having an adversary who can add or remove agents with different opinion. However, all agents are assumed already to have an estimate of $\log n$ that remains untouched. Thus it is not straightforward to use their protocol to solve our problem of obtaining this estimate.

2 Definitions and Notation

A *population protocol* is a pair $\mathcal{P} = (\Lambda, \Delta)$, where Λ is a finite set of *states*, and $\Delta \subseteq (\Lambda \times \Lambda) \times (\Lambda \times \Lambda)$ is the *transition relation*. (Often this is defined as a function $\delta : \Lambda \times \Lambda \rightarrow \Lambda \times \Lambda$, but we allow randomized transitions, where the same pair of inputs can randomly choose among multiple outputs.)

A *configuration* \mathbf{c} of a population protocol is a multiset over Λ of size n , giving the states of the n agents in the population. For a state $s \in \Lambda$, we write $\mathbf{c}(s)$ to denote the count of agents in state s . A *transition* is a 4-tuple, written $\alpha : r_1, r_2 \rightarrow p_1, p_2$, such that $((r_1, r_2), (p_1, p_2)) \in \Delta$. If an agent in state r_1 interacts with an agent in state r_2 , then they can change states to p_1 and p_2 . This notation omits explicit probabilities; our main protocol's transitions can be implemented so as to always have either one or two possible outputs for any input pair, with probability 1/2 of each output in the latter case.² For every pair of states r_1, r_2 without an explicitly listed transition $r_1, r_2 \rightarrow p_1, p_2$, there is an implicit *null* transition $r_1, r_2 \rightarrow r_1, r_2$ in which the agents interact but do not change state. For our main protocol, we specify transitions formally with pseudocode that indicate how agents alter each independent field in their state. We say a configuration \mathbf{d} is *reachable* from a configuration \mathbf{c} if applying 0 or more transitions to \mathbf{c} results in \mathbf{d} .

When discussing random events in a protocol of population size n , we say event E happens *with high probability* if $\Pr[\neg E] = O(n^{-c})$, where c is a constant that depends on our choice of parameters in the protocol, where c can be made arbitrarily large by changing the parameters.

² For the purpose of representation, we make an exception in our protocol, when we show agents generate a geometric random variable in one line (see Protocol 6). However, we can assume a geometric random variable is generated through $O(\log n)$ consecutive interactions with each selecting out of two possible outputs (**H** or **T**).

For concreteness, we will write a particular polynomial probability such as $O(n^{-2})$, but in each case we could tune some parameter (say, increasing the time complexity by a constant factor) to increase the polynomial's exponent.

To measure time we count the total number of interactions (including null transitions such as $a, b \rightarrow a, b$ in which the agents interact but do not change state), and divide by the number of agents n .

In a *uniform* protocol (such as the main one of this paper), the transitions are independent from the population size n (see [21] for a formal definition). In other words, a single protocol computes the output correctly when applied on any population size. In contrast, in a nonuniform protocol different transitions are applied for different population sizes.

A protocol *stably* solves a problem if the agents eventually reach a correct configuration with probability 1, and no subsequent interactions can move the agents to an incorrect configuration; i.e., the configuration is *stable*. A population protocol is self-stabilizing if from any initial configuration, the agents stably solve the problem.

3 Dynamic Size Counting

In a population of size n , define $C(n, \epsilon_1, \epsilon_2)$ to be the set of correct configurations \mathbf{c} such that every agent u in \mathbf{c} obeys $\epsilon_1 \log n < u.\text{estimate} < \epsilon_2 \log n$. Let t_h be any time bound. Moreover, we define $L(n, t_h) \subset C(n, \epsilon_1, \epsilon_2)$ the subset of correct configurations such that as the expected time for protocol P starting from a configuration $\mathbf{l} \in L(n, t_h)$ to stay in $C(n, \epsilon_1, \epsilon_2)$ is at least $t_h(n)$.

► **Definition 3.1.** Let n_{prev} and n_{next} denote the previous and next population size. A protocol P solves the dynamic size counting problem if there are $\epsilon_1, \epsilon_2 > 0$, called the accuracy, such that if the population size changes from n_{prev} to n_{next} , the protocol reaches a configuration \mathbf{l} in $L(n_{\text{next}}, t_h)$ with high probability. The time needed to do this is called the convergence time. Moreover, t_h , the time that the population stays in $C(n_{\text{next}}, \epsilon_1, \epsilon_2)$, is called the holding time.

A population protocol is $(t_c(n), t_h(n))$ -loosely stabilizing if starting from any initial configuration, the agents reach a correct configuration in $t_c(n)$ time and stay in the correct configuration for additional $t_h(n)$ time [33,34]. In contrast to self-stabilizing [7,17], subsequent interactions can move the agents to an incorrect configuration; however, the agents recover quickly from an incorrect configuration.

Given any starting configuration $\mathbf{s} \notin C(n, \epsilon_1, \epsilon_2)$ of size n , we define $f_c(\mathbf{s}, L(n, t_h))$ as the expected time to reach a correct configuration in $L(n, t_h)$.

► **Definition 3.2** ([34, Definition 2]). Let $t_c(n, M)$ and $t_h(n)$ be functions of n , the largest integer value M in the initial configuration \mathbf{s} , and the set of correct configuration $C(n, \epsilon_1, \epsilon_2)$. A protocol P is a $t_c(n, M), t_h(n), \epsilon_1, \epsilon_2$ loosely-stabilizing population size counting protocol if there exists a set $L(n, t_h) \subset C(n, \epsilon_1, \epsilon_2)$ of configurations satisfying:

For every n and every initial configuration $\mathbf{s} \notin C(n, \epsilon_1, \epsilon_2)$ of size n , $f_c(\mathbf{s}, L(n, t_h)) \leq t_c(n, M)$.

3.1 Basic properties of the dynamic size counting problem

We first observe that the key challenge in dynamic size counting is that the adversary may remove agents. If the adversary can only add agents, the problem is straightforward to solve with optimal convergence and holding times.

► **Observation 3.3.** *Suppose the adversary in the dynamic size counting problem only adds agents. Then there is a protocol solving dynamic size counting with $O(\log n)$ convergence time (in expectation and with probability $\geq 1 - O(1/n)$) and infinite holding time.*

Proof. Each agent in the initial state s generates a geometric random variable. After the last time that the adversary adds agents, resulting in n total agents, exactly n geometric random variables will have been generated. Agents propagate the maximum by epidemic using transition $a, b \rightarrow \max(a, b), \max(a, b)$, taking $3 \ln n$ time to reach all agents with probability $\geq 1 - \frac{1}{n^2}$ [17, Corollary 2.8]. The maximum of n i.i.d. geometric random variables is in the range $[\log n - \log \ln n, 2 \log n]$ with probability $\geq 1 - \frac{1}{n}$ [18, Lemma D.7]. ◀

In contrast, if the adversary can *remove* agents, then even if it is guaranteed to do this exactly once, no protocol can be stabilizing, i.e., have infinite holding time.

► **Observation 3.4.** *Suppose the adversary in the dynamic size counting problem will remove agents exactly once. Then any protocol solving the problem has finite holding time.*

Proof. Suppose otherwise. Let the initial population size be n and the later size be $n' < n$. The protocol must handle the case where the adversary *never* removes agents, since in population size n this is equivalent to an adversary who starts with $n + 1$ agents and immediately removes one of them. Thus if the adversary waits sufficiently long before the removal, then all agents stabilize to output $k = \Theta(\log n)$. In other words, no sequence of transitions can alter the value, including transitions occurring only among any subpopulation of size n' . So after the adversary removes $n - n'$ agents, the remaining n' agents are unable to alter the output k , a contradiction if n' is sufficiently small compared to n such that the output k is not a correct estimate for a population of size n' . ◀

Recall that we define M as the largest integer value the agents stored in the starting configuration \mathbf{s} . Lemma 3.5 shows that the dynamic size counting problem is equivalent to the loosely-stabilizing counting problem. Due to this equivalence, our correctness proofs will use the loosely-stabilizing characterization. The proof is given in the full version [20].

► **Lemma 3.5.** *A protocol solves the dynamic size counting problem with convergence time $t_c(n, M)$ and holding time $t_h(n)$ if and only if it solves the loosely-stabilizing counting problem with convergence time $t_c(n, M)$ and holding time $t_h(n)$.*

Proof sketch. Any states present in an adversarially prepared configuration \mathbf{c} will be produced in large quantities from any sufficiently large initial configuration of all initialized states s [18, Lemma 4.2]. The dynamic size adversary can then remove agents to result in \mathbf{c} , which the protocol must handle, showing it can handle an arbitrary initial configuration. ◀

3.2 High-level overview of dynamic size counting protocol

This section briefly describes our protocol for solving the dynamic size counting, defined formally in Section 3.3. By Lemma 3.5, it suffices to design a protocol solving the loosely-stabilizing counting problem for a fixed population size n . Our protocol uses the “detection” protocol of [3]. Consider a subset of states designated as a “source”. A detection protocol alerts all agents whether a source state is present in the population.

In Protocol 1, the population maintains several dynamic *groups*, with the agent’s group stored as a positive integer field `group`. The `group` values are not fixed: each agent changes its `group` field on every interaction, with equal probability either incrementing `group` or setting it to 1. We show that, no matter the initial group values, after $O(\log n)$ time the group values will be in the range $[1, 8 \log n]$ WHP. Furthermore, the distribution of `group` values is very close to that of n i.i.d. geometric random variables, in the sense that each agent’s `group` value is independent of every other, with expected $n/2^i$ agents having `group` = i if each agent has had at least i interactions.³

The agents store an array of “signal” integers in their `signals` field to track the existing `group` values in the population. Each agent in the i ’th group is responsible for *boosting* the signal associated with i . The goal is to have `signals`[i] > 0 for all agents if and only if some agent has `group` = i .

The detection protocol of [3], explained below, provides a technique for agents to know which groups are still present. Once a signal for group k fades out, the agents speculate that there is no agent with `group` = k . Depending on the current value stored as `estimate` in agents’ memory and the value k , this might cause re-calculating the population size. The agents are constantly checking for the changes in the `signals`. They re-compute `estimate` once there is a large gap between `estimate` and the first group i with `signals`[i] = 0. We call i the *first missing value* (stored in the field FMV).

The `signals` array is updated as follows. An agent with `group` = k sets `signals`[k] to its maximum possible value ($3k + 1$); we call this *boosting*. Other groups k are updated between two agents u, v with `u.signals`[k] = a and `v.signals`[k] = b via *propagation* transitions that set both agent’s `signals`[k] to $\max(a - 1, b - 1, 0)$. The paper [3] used a nonuniform protocol where each agent *already* has an estimate of $\log n$. They prove that if the state being detected (in our case, a state with `group` = k) is *absent* and the current maximum signal is c , then all agents will have signal 0 within $\Theta(c)$ time. However, if the state being detected is present, then the boosting transitions (occurring every $O(1)$ units of parallel time on average in the worst case that its count is only 1) will keep the signal positive in all agents with high probability. For this to hold, the maximum value set during boosting must be $\Omega(\log n)$; the nonuniform protocol of [3] uses its estimate of $\log n$ for this purpose.

Crucially, our protocol associates smaller maximum signal values to smaller group values (so many are much smaller than $\log n$), to ensure that a signal does not take abnormally long to get to 0 when its associated group value is missing. Otherwise, if we set each signal value to $\Omega(\log n)$ (based on the agent’s current estimate of $\log n$) during boosting, then it would take time proportional to `estimate` (which could be much larger than the actual value of $\log n$) to detect the absence of a `group` value. Thus it is critical that we provide a novel analysis of the detection protocol, showing that the signals for smaller group values $k \ll \log n$ remain present with high probability. This requires arguing that the boosting reactions for such smaller values are happening with sufficiently higher frequency, due to the higher count of agents with `group` = k , compensating for the smaller boosting signal values they use.

3.3 Formal description of loosely-stabilizing counting protocol

The DynamicCounting protocol (Protocol 1) divides agents among several groups via the UpdateGroup subprotocol. The agents update their `group` from i to $i + 1$ with probability $1/2$ or reset to group 1 with probability $1/2$. The number of agents at each group and the

³ The difference is that a geometric random variable G obeys $\Pr[G = j] = 1/2^j$ for all $j \in \mathbb{N}^+$, but after i interactions an agent u can increment `u.group` by at most i , so $\Pr[u.\text{group} = j] = 0$ if $j \gg i$.

13:8 Dynamic Size Counting in Population Protocols

total number of groups are both random variables dynamically changing through time. We show that the total number of groups remains close to $\log n$ at all times with high probability.

The agents start with arbitrary (or even adversarial) **group** values but we show that WHP the set of **group** values will converge to $[1, 8 \log n]$ within $O(\log n)$ time. Additionally, each agent stores an array of $O(\log n)$ signal values in their **signals** field. The goal is to maintain positive values in the **signals**[i] if some agent has **group** = i . The agents store the index of the first **group** i with **signals**[i] = 0 in their **FMV** field. They use **FMV** as an approximation of $\log n$ and constantly compare it with their **estimate** value.

Depending on the **estimate** value stored in agents' memory, the agents maintain three main phases of computation:

NormalPhase: An agent stays in the NormalPhase as long as there is a small gap between **estimate** and **FMV**: $0.25 \cdot \text{estimate} \leq \text{FMV} \leq 2.5 \text{estimate}$.

WaitingPhase: An agent switches from NormalPhase to WaitingPhase if it sees a large gap between the **FMV** and **estimate**: $\text{FMV} \notin \{0.25 \cdot \text{estimate}, \dots, 2.5 \cdot \text{estimate}\}$. The purpose of WaitingPhase is to give enough time to the other agents so that by the end of the WaitingPhase for one agent, with high probability every other agent has also noticed the large gap between the **FMV** and **estimate** and entered WaitingPhase.

UpdatingPhase: During the UpdatingPhase, every agent uses a new geometric random variable and propagates the maximum by epidemic. We set WaitingPhase long enough so that with high probability when the first agent switches to the UpdatingPhase, the rest of the population are all in WaitingPhase. By the end of UpdatingPhase, every agent switches back to NormalPhase.

Below we explain each subprotocol in more detail.

■ Algorithm 1 DynamicCounting(u, v).

```
for agent  $\in$  { $u, v$ } do
    UpdateGroup(agent)
SignalPropagation( $u, v$ )
for agent  $\in$  { $u, v$ } do
    UpdateMV(agent)
    SizeChecker(agent)
    if agent.phase  $\neq$  NormalPhase then
        TimerRoutine(agent)
PropagateMaxEst( $u, v$ )
for agent  $\in$  { $u, v$ } do
    if agent.phase = NormalPhase then
        agent.estimate  $\leftarrow$  agent.GRV
```

In every interaction, both sender and receiver update their group according to the rules of the UpdateGroup subprotocol. If we look at the distribution of the **group** values after $O(\log n)$ time, there are about $n/2$ agents in group 1, $n/4$ agents in group 2, and $n/2^i$ agents in group i (see Figure 1). Note that the number of agents in each group decreases exponentially. Still, we ensure that agents with larger **group** values use stronger signals to propagate, since there is less support for those groups.

To notify all agents about the set of all **group** values that are generated among the population, we use the detection protocol of [3] that is also used as a synchronization scheme in [12]. The agents store an integer for each **group** value that is generated by the population. The **signals** is an array of length $\Theta(\log n)$ such that a positive value in index i represents

Algorithm 2 UpdateGroup(u).

$$u.\text{group} \leftarrow \begin{cases} u.\text{group} + 1 & \text{with probability } 1/2 \\ 1 & \text{with probability } 1/2 \end{cases}$$

some agents in the population have generated $\text{group} = i$. Note that, as an agent updates its group , it boosts multiple signals based on its group value, e.g., an agent with $\text{group} = i$ helped boost all the indices $1, 2, 3, \dots, i$ of signals in its last i interactions. We use the SignalPropagation protocol to keep the signal of group i positive as long as some agents have generated $\text{group} = i$.

Algorithm 3 SignalPropagation(u, v).

▷ Boosting:
 $u.\text{signals}[u.\text{group}] \leftarrow (3 \cdot u.\text{group}) + 1$
 $v.\text{signals}[v.\text{group}] \leftarrow (3 \cdot v.\text{group}) + 1$
 ▷ Propagate signal:
for $i \in \{1, 2, \dots, \text{Max}(|u.\text{signals}|, |v.\text{signals}|)\}$ **do**
 $m \leftarrow \text{Max}(u.\text{signals}[i], v.\text{signals}[i])$
 $u.\text{signals}[i], v.\text{signals}[i] \leftarrow \text{Max}(0, m - 1)$

Regardless of the initial configuration, the distribution of group values changes immediately (in $O(\log n)$ time), but it might take more time for the signals to get updated. It takes $O(i)$ time for $\text{signals}[i]$ to hit zero. The larger the index i , $\text{signals}[i]$ leaves the population slower. Hence, the agents look at the *first missing signal* that they observe among the array of all signals.

Algorithm 4 UpdateMV(u).

▷ Find the first appearance of a zero in $u.\text{signals}$ beyond index $\lceil \log(u.\text{estimate}) \rceil$
 $s \leftarrow \lceil \log(u.\text{estimate}) \rceil$
 $u.\text{FMV} \leftarrow \min\{i \in [s, |u.\text{signals}|] \mid u.\text{signals}[i] = 0\}$

Once there is a large gap between the first missing group (FMV) and the agents' estimation of $\log n$ (estimate), each agent individually moves to a *waiting phase* and waits for other agents to catch the same gap between their estimate and FMV. Note that we time this phase as a function of FMV and not the estimate since the estimate is not valid anymore and might be much smaller or larger than the actual value of $\log n$.

Eventually, all agents will notice the large discrepancy between FMV and estimate and move to the *WaitingPhase*. The *WaitingPhase* is followed by the *UpdatingPhase* (explained in the *TimerRoutine*). In the *UpdatingPhase*, all agents generate one geometric random variable (stored in GRV) and propagate the maximum value. We assume the agents generate a geometric random variable in one line (line 4 in *Protocol 6*) for simplicity.⁴

Once the *UpdatingPhase* is completed, all agents will update their estimate to the maximum geometric random variable they have seen and switch to the *NormalPhase* again. Recall that the agents remain in the *NormalPhase* as long as their FMV and estimate are relatively close. They continue changing their group values and send group signals as described earlier.

⁴ Alternatively, the agents could generate a geometric random variable through $O(\log n)$ consecutive interactions, each selecting a random coin flip (**H** or **T**). In this alternative version, we should make the *WaitingPhase* longer.

13:10 Dynamic Size Counting in Population Protocols

■ Algorithm 5 SizeChecker(u).

```

if  $u.\text{phase} = \text{NormalPhase}$  and  $u.\text{FMV} \notin \{0.25 \cdot u.\text{estimate}, \dots, 2.5 \cdot u.\text{estimate}\}$  then
     $u.\text{phase} \leftarrow \text{WaitingPhase}$            ▷ Waiting for other agents to detect the size change

```

■ Algorithm 6 TimerRoutine(u).

```

 $u.\text{timer} \leftarrow u.\text{timer} + 1$ 
if  $u.\text{timer} > 12 \cdot u.\text{FMV}$  then
    if  $u.\text{phase} = \text{WaitingPhase}$  then
         $u.\text{GRV} \leftarrow$  a new geometric random variable
         $u.\text{timer} \leftarrow 0$ ,  $u.\text{phase} \leftarrow \text{UpdatingPhase}$ 
    if  $u.\text{phase} = \text{UpdatingPhase}$  then
         $u.\text{estimate} \leftarrow u.\text{GRV}$ 
         $u.\text{timer} \leftarrow 0$ ,  $u.\text{phase} \leftarrow \text{NormalPhase}$ 

```

Intuitively, for each **group** value, about $n/2^i$ agents will hold **group** = i , and boost **signals**[i] by setting it to the max = $\Theta(i)$. As the value of i grows, the number of agents with **group** = i decreases, but their signals get stronger since the agents enhance a **group** signal i proportional to i . In a normal run of the protocol, the agents expect to have positive values in **signals**[i] for **group** values between $[\log \ln n, \log n]$.

4 Analysis of Dynamic Counting Protocol

4.1 Bound on the group values

Recall that the agents calculate a dynamic **group** value by following the rules of Protocol 2. As described in this protocol, the agents either move to the next **group** or return to **group** 1 with probability $1/2$.⁵

In this part, we analyze the distribution of **group** values. Note that the **group** values are rather chaotic at the beginning of the protocol since the agents might start holding any arbitrary **group** values that are much larger than $\log n$. However, after all agents reset back to **group** = 1, we can show for each **group** = k , $\Pr[\text{group} = k] \approx \frac{1}{2^k}$.

In the rest of this section, we assume the initialized setting for simplicity. Later, we show how we can generalize our results to any arbitrary initial configuration. We define $G_{u,t}$ as the **group** value of agent u at time t and $I(t, u)$ to represent the number of interactions involving this agent by the time t . Note that with this definition, $G_{u,t}$ is equal to k (for $k < I(t, u)$) if and only if agent u generates the sequence of $[HTTT \dots T]$ (**H** followed by $k - 1$ **T**s) during its last k interactions. Thus, we have:

$$\forall k \in \mathbb{N}, \quad 1 \leq k < I(t, u) : \Pr[G_{u,t} = k] = \frac{1}{2^k} \quad (1)$$

With this definition $G_{u,t}$ is undefined for any agents that has not generated **H** yet. In other words, the values $G_{u,t}$ are “close to geometric” in the sense that they are independent and have probability equal to a geometric random variable on all values $k < I(t, u)$.

⁵ The truncated version of this Markov chain (mapping all states $k + 1, k + 2, \dots$ to $k + 1$) is also known as the “winning streak” [30].

■ **Algorithm 7** PropagateMaxEst(u, v).

if $u.\text{phase} = v.\text{phase}$ & $u.\text{phase} \neq \text{WaitingPhase}$ then
 $u.\text{GRV}, v.\text{GRV} \leftarrow \max(u.\text{GRV}, v.\text{GRV})$

► **Observation 4.1.** For agents u_1, u_2, \dots, u_n , and the values $k_i < I(t, u_i)$, for $1 \leq i \leq n$:

$$\Pr [G_{u_1, t} = k_1, G_{u_2, t} = k_2, \dots, G_{u_n, t} = k_n] = \prod_{i=1}^n \Pr [G_{u_i, t} = k_i]$$

Next we bound the maximum **group** value that has been generated by any agent. Let $M_t = \max_{u \in \mathcal{A}} G_{u, t}$ be the maximum value of $G_{u, t}$ across the population at time t . A proof of the following lemma appears in the full version [20].

► **Lemma 4.2.** Let $c \geq 2$ and let t be a time such that all agents have at least $c \log n$ interactions. In a population of size n , $\frac{1}{d} \log n \leq M_t$ with probability at least $1 - \exp(-n^{1-1/d})$ and $M_t < c \log n$ with probability at least $1 - n^{1-c}$.

Note that the maximum **group** value has a large variance. However, we can prove a tight bound for the first **group** value with no support; since to have $\text{FMV} = k$, for all values i that are less than k , $\exists u \in \mathcal{A}$ such that $u.\text{group} = i$.

So, we analyze the bounds for the first **group** value with no support, i.e., the value $\min\{k \in \mathbb{N}^+ \mid (\forall u \in \mathcal{A}) u.\text{group} \neq k\}$. Considering n i.i.d. geometric random variables, the *first missing value* to be the smallest integer not appearing among the random variables. The first missing value has been studied in the literature [28, 29, 32] as the “the first empty urn” (see also “probabilistic counting” [23]) but for simplicity we use a loose bound for our analysis. The proof appears in the full version [20].

► **Lemma 4.3.** Let $\delta > 0$, $0 < \epsilon < 1$ and let t be a time such that all agents have at least $(1 + \delta) \log n$ interactions. Define $\text{FMV}_t = \min\{k \in \mathbb{N} \mid (\forall u \in \mathcal{A}) u.\text{group} \neq k\}$ at time t . Then, $\text{FMV}_t > (1 - \epsilon) \log n$ with probability at least $(1 - \epsilon) \log(n) \cdot \exp(-n^\epsilon)$ and $\text{FMV}_t \leq (1 + \delta) \log n$ with probability at least $1 - \left(\frac{1}{n^{\delta/2}}\right)^{(2+\delta) \log n}$.

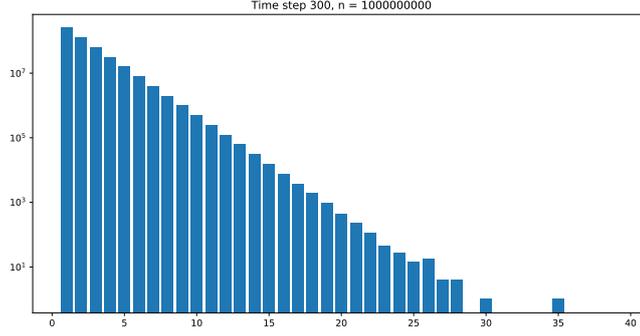
4.2 Distribution of the groups

So far, we have proved bounds on the existing **group** values. However, in general, we need to show that at a given time $t = \Omega(\log n)$, there are about $\frac{n}{2^k}$ agents having $\text{group} = k$ WHP. The following lemma gives us a lower and upper bound for the number of agents in each **group**:

► **Lemma 4.4.** Let $c \geq 2$, $0 < \epsilon < 1$, $0 \leq \delta \leq 1$, and let t be a time such that all agents have at least $c \log n$ interactions. Let $1 \leq k \leq (1 - \epsilon) \log n$, then, the number of agents who hold $\text{group} = k$, is at least $L_k = (1 - \delta) \frac{n}{2^k}$ with probability at least $1 - \exp\left(-\frac{\delta^2 \cdot n^\epsilon}{2}\right)$ and at most $U_k = (1 + \delta) \frac{n}{2^k}$ with probability at least $1 - \exp\left(-\frac{\delta^2 \cdot n^\epsilon}{3}\right)$.

Proof sketch. The fraction of agents with $\text{group} = k$ is equal to the fraction of heads in of a binomial distribution $B(n, 2^{-k})$ with $\mu = \frac{n}{2^k}$, so the Chernoff bound applies. A complete proof is given in the full version [20]. ◀

The following theorem summarizes what we will use later about the distribution of the **group** values and the number of agents residing in each **group** at time t .



■ **Figure 1** Showing the distribution of `group` values after 300 parallel-time in a population of size $n = 10^9$. The x-axis indicates the different `group` values while the y-axis indicates the number of agents in each group. Note that, we are using log-scale for the y-axis. In this snapshot of the population, $\text{FMV} = 29$. Even though, the maximum `group` value is 35 and is much larger than FMV .

► **Theorem 4.5.** Fix a time $t \geq d \ln n$ for $d > 30$, let M_t^* and FMV_t be the maximum `group` value and the FMV at this time respectively. Then,

- $0.9 \log n \leq M_t^* < 0.1d \log n$ with probability at least $1 - 2 \cdot n^{1-d/10} - 2 \cdot n^{1-\frac{2d}{3}}$.
- $0.9 \log n \leq \text{FMV}_t < 3 \log n$ with probability at least $1 - 4 \cdot n^{1-\frac{2d}{3}}$.
- The number of agents who hold `group` = k for $1 \leq k \leq 0.9 \log n$, is in $\left[\frac{3 \cdot n}{2^{k+2}}, \frac{5 \cdot n}{2^{k+2}}\right]$ with probability at least $1 - 4 \cdot n^{1-\frac{2d}{3}}$.

4.3 Group detection

In the previous section, we show that the set of present `group` values among the population will quickly (in $O(\log n)$ time) enter a small interval of values $([1, 8 \cdot \log n])$ consistent with the population size. In this section, we will prove the following:

- The agents *agree* about the presence of `group` values in $[\log \ln n, 0.9 \log n]$ after $O(\log n)$ time WHP.
- For a non-existing `group` value i , each agent will have `signals`[i] = 0 in $O(i + \log n)$ time WHP.

We designed Protocol 3 such that each agent in the i 'th `group` *boosts* the associated signal value by setting `signals`[i] = B_i (recall $B_i = \Theta(i)$). We will show by having at least L_i agents boosting `signals`[i], the whole population learns about the existence of the i 'th `group` in $O(\log n)$ time with high probability. Intuitively, although `signals`[i] starts lower than `signals`[j] for $i < j$, so potentially dies out more quickly, it is also boosted more often since more agents have `group` value i . Concretely, with L_i agents responsible to boost signal i , and for all indices $\log \ln n < i < 0.9 \log n$ in the `signals` of the agents, $\Pr[\text{signals}[i] = 0] < \exp\left(-\frac{2^{B_i}}{n/L_i}\right)$.

Intuitively, the next lemma shows that if the `group` values are distributed as in Lemma 4.4, then the whole population will learn about all the present `group` values above $\log \ln n$ within $O(\log n)$ time. Note that $\Pr[u.\text{signals}[i] = j]$ is the probability that the agent u has value j in the i 'th index of its `signals`. The following lemma is a restatement from [3, Section 5.1]. The proof appears in the full version [20].

► **Lemma 4.6.** In the execution of Protocol 3, suppose that for each `group` value $\log \ln n < i < 0.9 \log n$, at least A_i agents hold `group` = i . For every agent $u \in \mathcal{A}$ let $u.\text{signals}[i] = r_i$ when $u.\text{group} = i$. Assuming each agent has at least r_i interactions, then for a fixed agent u and index i , $\Pr[u.\text{signals}[i] = 0] \leq \left(1 - \frac{A_i}{n}\right)^{2^{r_i-1}}$.

To use the previous lemma, we need to make sure that the agents wait for sufficiently long time such that each agent has at least r_i interactions. The next corollary uses Lemma 4.6 to derive bounds for the entire protocol using bounds from Lemma 4.4 for the distribution of the `group` values. Also, Corollary 4.7 takes a union bound over *all* agents and group values i , and uses the concrete value $r_i = B_i = 3 \cdot i + 1$ used in our protocol. The proof appears in the full version [20].

► **Corollary 4.7.** *For all $i > 0$ and for every agent $u \in \mathcal{A}$, assuming $B_i = 3 \cdot i + 1$ let $u.\text{signals}[i] = B_i$ if $u.\text{group} = i$. Suppose that for each group value $\log \ln n < i < 0.9 \log n$, at least L_i agents hold $\text{group} = i$. Let $\beta \geq 8$; then after $\beta \log n$ time, we have:*

$$\Pr[(\exists u \in \mathcal{A})(\exists i \in \{\log \ln n, \dots, 0.9 \log n\}) u.\text{signals}[i] = 0] \leq 2 \cdot n^{1-0.9\beta}$$

Finally, we show that when there is no agent holding $\text{group} = i$, then `signals`[i] will become zero in all agents “quickly” with an arbitrarily large probability. To be precise, with no agent boosting signal i , $\Pr[u.\text{signals}[i] = 0] \geq 1 - n^{-\alpha}$ within $\Theta(B_i + \alpha \ln n)$ time WHP in which B_i is the maximum value for signal i . The lemma is a restatement from [17, Lemma 3.3] and [3, Lemma 1].

► **Lemma 4.8.** *For every agent $u \in \mathcal{A}$ let $u.\text{signals}[i] = B_i$ when $u.\text{group} = i$. Assume that no agent sets its `group` to i from this point on. Then for all $\alpha \geq 1$, all agents will have $\text{signals}[i] = 0$ after $3n \ln(n^\alpha \cdot 3^{B_i})$ interactions with probability at least $1 - n^{-\alpha}$.*

Proof. Set $t = 3n \ln(n^\alpha \cdot 3^{B_i})$ and $R_{max} = B_i$ in the proof of [17, Lemma 3.3]. ◀

4.4 Dynamic size counting protocol analysis

Recall that `estimate` denote the estimate of $\log n$ in agents’ memory, and n is the true population size. In the previous section, we show that the set of present `group` values among the population will quickly (in $O(\log n)$ time) enter a small sub-interval of consecutive values in $[1, 3 \cdot \log n]$ consistent with the population size. This section will show that the `group` values will remain in that interval (with high probability for polynomial time). Moreover, the following two lemmas show how the agents update their `estimate` if it is far from $\log n$. The proofs appear in the full version [20].

Assuming the agents’ `estimate` is much smaller than $\log n$, the next lemma shows that all the agents will notice the large gap between `estimate` and `FMV`. Hence, they will re-calculate their population size estimate.

► **Lemma 4.9.** *Let $M = \max_{u \in \mathcal{A}} u.\text{estimate}$. Assuming $M \leq 0.22 \log n$, then the whole population will enter `WaitingPhase` in $O(\log n)$ time with probability at least $1 - O(n^{-2})$.*

For the other direction, assume the population size estimate in agents’ memory is much larger than $\log n$. We prove in the following lemma that all the agents will notice the large gap between `estimate` and `FMV`. Hence, they will re-calculate their population size estimation.

Note that in Corollary 4.7, we proved for all `group` values i for $\log \ln n \geq i$, the `signals`[i] will have a positive value in $O(\log n)$ time. However, we could not prove the same bound for values less than $\log \ln n$. So, inevitably the agents ignore their `signals` for values that are less than $\log \ln n$. Since the agents have no access to the value of $\log n$, they have to use `estimate` as an approximation of $\log n$. Thus, they ignore indices that are less than $\log M$ in `signals`: making `FMV` a function of $\max(\log M, \log n)$. For example, if the true population size is n but $M > 2^n$, then the agents should ignore the appearance of a zero in their `signals` for all indices i that are $\leq \log(M) = n$. The correct `FMV` happens at index

13:14 Dynamic Size Counting in Population Protocols

$j = \Theta(\log n)$, but the agents stay in the NormalPhase as long as `signals[i]` for $i \geq n$ are positive. In this scenario, it takes $O(n)$ time for the agents to switch to WaitingPhase since for each `signals[i]`, it takes $O(i)$ time to hit zero.

This scenario is inevitable with our current detection scheme since for indices i that are less than $\log \ln n$, the event of `signals[i] = 0` happens frequently.

► **Lemma 4.10.** *Let $M = \max_{u \in \mathcal{A}} u.\text{estimate}$. Assuming $M \geq 7.5 \cdot \log n$, then the whole population will enter WaitingPhase in $O(\log n + \log M)$ time with probability at least $1 - O(n^{-2})$.*

In the next theorem (full proof given in [20]), we will show once there is a large gap between the maximum `estimate` among the population and the true value of $\log n$, the agents update their estimate in $O(\log n + \log M)$ time.

► **Theorem 4.11.** *Let $M = \max_{u \in \mathcal{A}} u.\text{estimate}$. Assuming `estimate` $\geq 7.5 \log n$ or `estimate` $\leq 0.2 \log n$, then every agent replaces its `estimate` with a new value that is in $[\log n - \log \ln n, 2 \log n]$ with probability $1 - O(1/n)$ in $O(\log n + \log M)$ time.*

Proof sketch. By Lemmas 4.9 and 4.10, once an agent notices the large gap between `estimate` and FMV, they switch to WaitingPhase. We set WaitingPhase long enough so when the first agent moves to UpdatingPhase, there is no agent left in the NormalPhase. Thus, they all re-generate a new geometric random variable and store the maximum as their `estimate`. ◀

In the full version of the paper [20], we show that the holding time of our protocol is polynomial. For the state complexity, recall that the adversary can initialize agents with large integer values to arbitrarily increase memory usage. Therefore, we should calculate the agents' memory concerning the fields defined in our protocol and the value that the adversary can set in them. Thus, we use s as the largest integer value that the adversary set in the agents' memory.

► **Theorem 4.12.** *Let $M = \max u.\text{estimate}$ for all $u \in \mathcal{A}$. There is a uniform leaderless loosely-stabilizing population protocol that WHP:*

1. *If $M > 7.5 \log n$ or $M < 0.2 \log n$ reaches to a configuration with all agents set their `estimate` with a value in $[\log n - \log \ln n, 2 \log n]$ in $O(\log n + \log M)$ parallel time.*
2. *If $0.75 \log n < M < 2.25 \log n$, then the agents hold a stable `estimate` during the following $O(n^{15})$ parallel time.*
3. *Assuming for every agent $u \in \mathcal{A}$, $\max(u.\text{estimate}, u.\text{GRV}, u.\text{group}, u.\text{signals.size}()) < s$ in the initial configuration, then the protocol uses $O(\log^2(s) + \log n \log \log n)$ bits per agent.*

4.5 Space optimization

In this section, we explain how to reduce the space complexity of the protocol from $O(\log^2(s) + \log n \log \log n)$ to $O(\log^2(s) + (\log \log n)^2)$ bits per agent.

In Protocol 1, the agents keep track of all the present `group` values using an array of size $O(\log n)$ (stored in `signals`) by mapping every `group = i` to `signals[i]`. We can reduce the space complexity of the protocol by reducing the `signals`' size. Let the agents map a `group = i` to `signals[$\lceil \log i \rceil$]`. So, instead of monitoring all $O(\log n)$ group values, they keep $O(\log \log n)$ indices in their `signals`. Thus, reducing the space complexity to $O(\log^2(s) + (\log \log n)^2)$ bits per agent.

Recall that in Protocol 1, there are $\approx \frac{n}{2^i}$ agents with `group` = i for $i \leq 0.9 \log n$ that help keep `signals`[i] positive. However, with this technique, there will be $\approx \sum_{i=2^j}^{2^{j+1}} \frac{n}{2^i}$ agents that are helping `signals`[i] to stay positive. So, every lemma in Section 4.3 about Protocol 3 holds. Finally, we update Protocol 5 so that the agents compare their `estimate` with 2^{LFMV} in which LFMV is the smallest index $i > \log \log M$ such that `signals`[i] = 0. On the negative side of this optimization, we get a less sensitive protocol with respect to the gap between agents' `estimate` and $\log n$.

► **Theorem 4.13.** *Let $M = \max u.\text{estimate}$ for all $u \in \mathcal{A}$. There is a uniform leaderless loosely-stabilizing population protocol that WHP:*

1. *If $M > 15 \log n$ or $M < 0.1 \log n$ reaches to a configuration with all agents set their `estimate` with a value in $[\log n - \log \ln n, 2 \log n]$ in $O(\log n + \log M)$ parallel time.*
2. *If $0.75 \log n < M < 2.17 \log n$, then the agents hold a stable `estimate` during the following $O(n^{15})$ parallel time.*
3. *Assuming for every agent $u \in \mathcal{A}$, $\max(u.\text{estimate}, u.\text{GRV}, u.\text{group}, u.\text{signals.size}()) < s$ in the initial configuration, then the protocol uses $O(\log^2(s) + (\log \log n)^2)$ bits per agent.*

5 Conclusion and open problems

In this paper, we introduced the dynamic size counting problem. Assuming an adversary who can add or remove agents, the agents must update their `estimate` according to the changes in the population size. There are several open questions related to this problem.

Reducing convergence time. Our protocol's convergence time depends on both the previous (n_{prev}) and next (n_{next}) population sizes, though exponentially less on the former: $O(\log n_{\text{next}} + \log \log n_{\text{prev}})$. Is there a protocol with optimal convergence time $O(\log n_{\text{next}})$?

Increasing holding time. Observation 3.4 states that the holding time must be finite, but it is likely that much longer holding times than $\Omega(n^c)$ for constant c are achievable. For the loosely-stabilizing leader election problem, there is a provable tradeoff in the sense that the holding time is at most exponential in the convergence time [26, 36]. Does a similar tradeoff hold for the dynamic size counting problem?

Reducing space. Our main protocol uses $O(s + (\log n)^{\log n})$ states (equivalent to $O(\log^2(s) + \log n \log \log n)$ bits). In Section 4.5, we showed how we can reduce the state complexity of our protocol to $o(n^\epsilon)$ (equivalent to $O(\log^2(s) + (\log \log n)^2)$ bits) by mapping more than one `group` to each index of the `signals`. With this trick, we reduce the size of the `signals` from $O(\log n)$ to $O(\log \log n)$. Another interesting idea is to replace our $O(\log n)$ detection scheme to $O(1)$ detection protocol of [22] which puts a constant threshold on the values stored in each index. So, it may be possible to reduce the space complexity even more to $O(c^{O(\log n)})$ (with all $O(\log n)$ indices present) or $O(c^{O(\log \log n)}) = \text{polylog}(n)$ (using our optimization technique to have $O(\log \log n)$ indices in the `signals`).

However, the current protocol of [22] has a one-sided error that makes it hard to compose with our protocol. With probability $\epsilon > 0$, the agents might say signal i has disappeared even though there exists agents with `group` = i in the population.

Additionally, in the presence of a uniform self-stabilizing synchronization scheme, one could think of consecutive rounds of independent size computation. The agents update their output if the new computed population size drastically differs from the previously computed

population size. Note that the self-stabilizing clock must be independent of the population size since we allow the adversary to change the value of $\log n$ by adding or removing agents. To the best of our knowledge, there is no such synchronization scheme available to population protocols.

References

- 1 Dan Alistarh, James Aspnes, David Eisenstat, Rati Gelashvili, and Ronald L Rivest. Time-space trade-offs in population protocols. In *SODA 2017: Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2560–2579. SIAM, 2017.
- 2 Dan Alistarh, James Aspnes, and Rati Gelashvili. Space-optimal majority in population protocols. In *SODA 2018: Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2221–2239. SIAM, 2018.
- 3 Dan Alistarh, Bartłomiej Dudek, Adrian Kosowski, David Soloveichik, and Przemysław Uznański. Robust detection in leak-prone population protocols. In *DNA Computing and Molecular Programming*, pages 155–171. Springer International Publishing, 2017.
- 4 Dan Alistarh and Rati Gelashvili. Polylogarithmic-time leader election in population protocols. In *ICALP 2015: Proceedings, Part II, of the 42nd International Colloquium on Automata, Languages, and Programming - Volume 9135*, pages 479–491. Springer-Verlag, 2015. doi:10.1007/978-3-662-47666-6_38.
- 5 Dan Alistarh, Martin Töpfer, and Przemysław Uznański. Fast and robust comparison in population protocols. In *PODC 2021: The ACM Symposium on Principles of Distributed Computing*, 2021.
- 6 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006.
- 7 Dana Angluin, James Aspnes, Michael J. Fischer, and Hong Jiang. Self-stabilizing population protocols. *ACM Trans. Auton. Adapt. Syst.*, 3(4):1–28, 2008. doi:10.1145/1452001.1452003.
- 8 James Aspnes, Joffroy Beauquier, Janna Burman, and Devan Sohler. Time and space optimal counting in population protocols. In *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*, volume 70, pages 13:1–13:17, 2017.
- 9 Joffroy Beauquier, Janna Burman, Simon Clavière, and Devan Sohler. Space-optimal counting in population protocols. In *Distributed Computing*, pages 631–646, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
- 10 Joffroy Beauquier, Julien Clement, Stephane Messika, Laurent Rosaz, and Brigitte Rozoy. Self-stabilizing counting in mobile sensor networks with a base station. In *Distributed Computing*, pages 63–76, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
- 11 Stav Ben-Nun, Tsvi Kopelowitz, Matan Kraus, and Ely Porat. An $O(\log^{3/2} n)$ parallel time population protocol for majority with $O(\log n)$ states. In *PODC 2020: Proceedings of the 39th Symposium on Principles of Distributed Computing*, pages 191–199. Association for Computing Machinery, 2020. doi:10.1145/3382734.3405747.
- 12 Petra Berenbrink, Felix Biermeier, Christopher Hahn, and Dominik Kaaser. Loosely-stabilizing phase clocks and the adaptive majority problem. In *SAND 2021: 1st Symposium on Algorithmic Foundations of Dynamic Networks*, 2021.
- 13 Petra Berenbrink, Robert Elsässer, Tom Friedetzky, Dominik Kaaser, Peter Kling, and Tomasz Radzik. A Population Protocol for Exact Majority with $O(\log^{5/3} n)$ Stabilization Time and $\Theta(\log n)$ States. In *32nd International Symposium on Distributed Computing (DISC 2018)*, volume 121 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:18, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.DISC.2018.10.

- 14 Petra Berenbrink, George Giakkoupis, and Peter Kling. Optimal time and space leader election in population protocols. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2020, pages 119–129, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3357713.3384312.
- 15 Petra Berenbrink, Dominik Kaaser, Peter Kling, and Lena Otterbach. Simple and efficient leader election. In *SOSA 2018: The 1st Symposium on Simplicity in Algorithms*, pages 9:1–9:11, 2018. doi:10.4230/OASIcs.SOSA.2018.9.
- 16 Andreas Bilke, Colin Cooper, Robert Elsässer, and Tomasz Radzik. Brief announcement: Population protocols for leader election and exact majority with $O(\log^2 n)$ states and $O(\log^2 n)$ convergence time. In *PODC 2017: Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 451–453. Association for Computing Machinery, 2017. doi:10.1145/3087801.3087858.
- 17 Janna Burman, Ho-Lin Chen, Hsueh-Ping Chen, David Doty, Thomas Nowak, Eric Severson, and Chuan Xu. Time-optimal self-stabilizing leader election in population protocols. In *PODC 2021: Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 33–44. ACM, 2021. doi:10.1145/3465084.3467898.
- 18 David Doty and Mahsa Eftekhari. Efficient size estimation and impossibility of termination in uniform dense population protocols. In *PODC 2019: Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 34–42. Association for Computing Machinery, 2019. doi:10.1145/3293611.3331627.
- 19 David Doty and Mahsa Eftekhari. A survey of size counting in population protocols. *Theoretical Computer Science*, 894:91–102, 2021. Building Bridges – Honoring Nataša Jonoska on the Occasion of Her 60th Birthday. doi:10.1016/j.tcs.2021.08.038.
- 20 David Doty and Mahsa Eftekhari. Dynamic size counting in population protocols, 2022. arXiv:2202.12864.
- 21 David Doty, Mahsa Eftekhari, Othon Michail, Paul G. Spirakis, and Michail Theofilatos. Brief Announcement: Exact Size Counting in Uniform Population Protocols in Nearly Logarithmic Time. In *32nd International Symposium on Distributed Computing (DISC 2018)*, volume 121 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 46:1–46:3, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.DISC.2018.46.
- 22 Bartłomiej Dudek and Adrian Kosowski. Universal protocols for information dissemination using emergent signals. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2018, pages 87–99, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3188745.3188818.
- 23 Philippe Flajolet and G Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31(2):182–209, 1985.
- 24 Leszek Gąsieniec, Grzegorz Stachowiak, and Przemysław Uznański. Almost logarithmic-time space optimal leader election in population protocols. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '19, pages 93–102. Association for Computing Machinery, 2019. doi:10.1145/3323165.3323178.
- 25 Shafi Goldwasser, Rafail Ostrovsky, Alessandra Scafuro, and Adam Sealfon. Population stability: regulating size in the presence of an adversary. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 397–406. ACM, 2018.
- 26 Taisuke Izumi. On space and time complexity of loosely-stabilizing leader election. In *Structural Information and Communication Complexity*, pages 299–312. Springer International Publishing, 2015.
- 27 Tomoko Izumi, Keigo Kinpara, Taisuke Izumi, and Koichi Wada. Space-efficient self-stabilizing counting population protocols on mobile sensor networks. *Theoretical Computer Science*, 552:99–108, 2014. doi:10.1016/j.tcs.2014.07.028.
- 28 Guy Louchard and Helmut Prodinger. The moments problem of extreme-value related distribution functions. *Algorithmica*, 2004.

- 29 Guy Louchard, Helmut Prodinger, and Mark Daniel Ward. The number of distinct values of some multiplicity in sequences of geometrically distributed random variables. In *Discrete Mathematics and Theoretical Computer Science*, pages 231–256. Discrete Mathematics and Theoretical Computer Science, 2005.
- 30 László Lovász and Peter Winkler. Reversal of markov chains and the forget time. *Combinatorics, Probability and Computing*, 7(2):189–204, 1998.
- 31 Yves Mocquard, Emmanuelle Anceaume, James Aspnes, Yann Busnel, and Bruno Sericola. Counting with population protocols. In *14th IEEE International Symposium on Network Computing and Applications*, pages 35–42, 2015.
- 32 Helmut Prodinger. Philippe flajolet’s early work in combinatorics. *arXiv preprint*, 2021. [arXiv:2103.15791](https://arxiv.org/abs/2103.15791).
- 33 Yuichi Sudo, Ryota Eguchi, Taisuke Izumi, and Toshimitsu Masuzawa. Time-optimal loosely-stabilizing leader election in population protocols. In *DISC 2021: The 35th International Symposium on Distributed Computing*, 2021.
- 34 Yuichi Sudo, Junya Nakamura, Yukiko Yamauchi, Fukuhito Ooshita, Hirotugu Kakugawa, and Toshimitsu Masuzawa. Loosely-stabilizing leader election in population protocol model. In *Structural Information and Communication Complexity*, pages 295–308, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- 35 Yuichi Sudo, Fukuhito Ooshita, Taisuke Izumi, Hirotugu Kakugawa, and Toshimitsu Masuzawa. Logarithmic expected-time leader election in population protocol model. In *Stabilization, Safety, and Security of Distributed Systems*, pages 323–337. Springer International Publishing, 2019.
- 36 Yuichi Sudo, Fukuhito Ooshita, Hirotugu Kakugawa, Toshimitsu Masuzawa, Ajoy K. Datta, and Lawrence L. Larmore. Loosely-Stabilizing Leader Election with Polylogarithmic Convergence Time. In *22nd International Conference on Principles of Distributed Systems (OPODIS 2018)*, volume 125 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.OPODIS.2018.30.

Simulating 3-Symbol Turing Machines with SIMD||DNA

David Doty   

University of California, Davis, CA, USA

Aaron Ong 

University of California, Davis, CA, USA

Abstract

SIMD||DNA [12] is a model of DNA strand displacement allowing parallel in-memory computation on DNA storage. We show how to simulate an arbitrary 3-symbol space-bounded Turing machine with a SIMD||DNA program, giving a more direct and efficient route to general-purpose information manipulation on DNA storage than the Rule 110 simulation of Wang, Chalk, and Soloveichik [12]. We also develop software [10] that can simulate SIMD||DNA programs and produce SVG figures.

2012 ACM Subject Classification Theory of computation → Models of computation

Keywords and phrases DNA storage, strand displacement, parallel computation

Digital Object Identifier 10.4230/LIPIcs.SAND.2022.14

Related Version *Full Version:* <https://arxiv.org/abs/2105.08559>

Supplementary Material *Software (Simulator Source Code):*

<https://github.com/UC-Davis-molecular-computing/simd-dna>

archived at [swh:1:dir:95e293f1f2de0273fcd5414a1f44848dd10685c](https://zenodo.org/record/6541414/files/95e293f1f2de0273fcd5414a1f44848dd10685c)

Funding The authors were supported by NSF grants 1900931 and 1844976.

1 Introduction

DNA storage typically encodes information in the choice of DNA sequences [1, 3, 7], so that reading and writing require expensive sequencing (reading DNA) and synthesis (writing DNA) steps. An alternative “nicked storage” scheme of Tabatabaei et al. [11] uses a single long strand called a *register*, with a fixed sequence. Information is stored in the choice of short complementary strands to bind to the register. This gives the potential to process the stored information using *DNA strand displacement* (see Figure 1), which reconfigures which DNA strands are bound, without changing their sequences. Thus manipulation of the stored information (i.e., computation) can potentially be done *in vitro* with simpler lab steps than DNA sequencing or synthesis.



Figure 1 DNA strand displacement (see [9] for more details). An input DNA strand (turquoise) binds to the short toehold region of a complementary strand (pink) and displaces the output strand (purple). The toehold region is so-called because, although too short to bind stably, it allows temporary binding of the input, giving it a “foot in the door” to begin the displacement process.

The SIMD||DNA model of Wang, Chalk, and Soloveichik [12] is an abstract model of such a system. It allows parallel in-memory computation on several copies of the register; each register may store different data. In the experimental implementation, each register strand is attached to a magnetic bead, enabling *elution*: washing away strands not bound to



© David Doty and Aaron Ong;

licensed under Creative Commons License CC-BY 4.0

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).

Editors: James Aspnes and Othon Michail; Article No. 14; pp. 14:1–14:15

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

a register, while keeping the registers (and their bound strands) in the solution due to their attachment to the bead. This motivates the “multi-stage” SIMD||DNA model of DNA strand displacement, which at a high level works as follows. Each stage is called an *instruction*, consisting of a set of strands to add to the solution. It is assumed that strand displacement reactions proceed until the solution reaches equilibrium, at which point all strands and complexes not attached to a register are washed away. The strands for the next instruction are then added. A key aspect of the model is that the wash step can constrain what strand displacement reactions are possible afterward, compared to “one-pot” strand displacement schemes that mix all strands from the start. This gives the SIMD||DNA model potentially more power than one-pot DNA strand displacement. Wang, Chalk, and Soloveichik [12] showed SIMD||DNA programs for binary counting and simulating cellular automata Rule 110, and Chen, Solanki, and Riedel [2] showed SIMD||DNA programs for sorting, shifting and searching in parallel. See Section 2 for a formal definition and [12] for more details and motivation for the model.

A major theoretical result of [12] is a SIMD||DNA program that simulates a space-bounded version of cellular automata Rule 110. When space is unbounded, Rule 110 is known to be efficiently Turing universal, i.e., able to simulate any single-tape Turing machine [4] with only a polynomial-time slowdown [5], though by an awkward indirect construction and encoding with very large constant factors. We show how to simulate an arbitrary 3-symbol space-bounded single-tape Turing machine *directly* with a SIMD||DNA program. Since custom manipulation of bits is much easier to program in a Turing machine than Rule 110, this gives a more direct, efficient, and conceptually simple method of general-purpose information processing on nicked DNA storage. Although we have not worked out the details, it seems likely that the construction can be extended straightforwardly to Turing machines with alphabet sizes larger than 3. However, it is straightforward to simulate a larger-alphabet Turing machine M with a 3-symbol Turing machine S , for example representing each of 16 non-blank symbols of M by 4 consecutive bits of S .

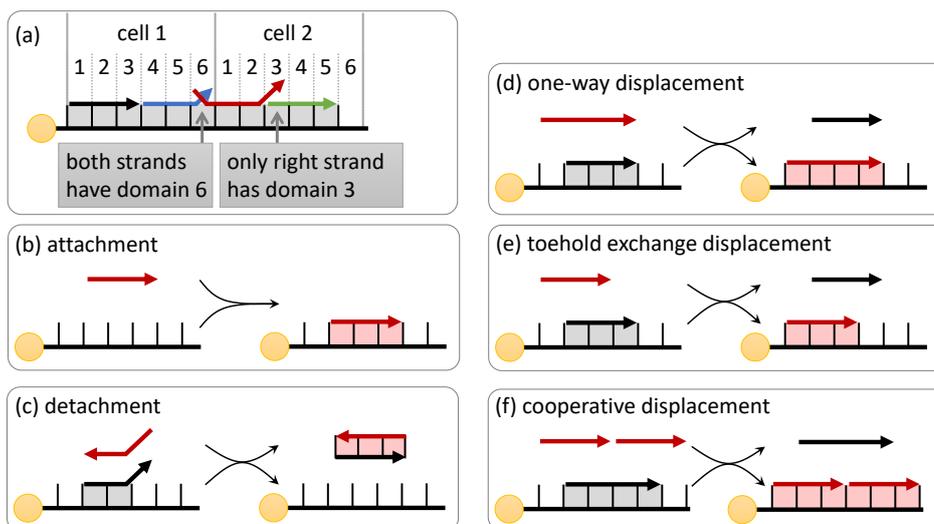
Our construction was designed and tested using software we developed [10] for simulating the SIMD||DNA model. It is able to take a description of an arbitrary SIMD||DNA program: a list of instructions, where each instruction is a set of DNA strands to add. It produces figures indicating visually how the steps work, both with text printed on the command line (for quickly testing ideas) and SVG figures, such as most of those in this paper.

2 Model

In this section we define the model of SIMD||DNA [12].

See Figure 2 for notational conventions in the SIMD||DNA model and an explanation of the basic strand displacement reactions. The register strand is on the bottom in each sub-figure, with a yellow round “magnetic bead” depicted on the left (bead not depicted in subsequent figures). A DNA strand has an orientation, with one end called the 5′ end and the other called the 3′ end; by convention strands are drawn as arrows with the arrowhead on the 3′ end. The register strand has its 3′ end on the left and 5′ end on the right. The model allows multiple registers to be present in solution at once, each possibly configured differently. However, it is assumed that register strands are sufficiently dilute that they do not interact with each other or with strands that have been displaced from other registers. Thus all figures depict only a single register and instruction strands that interact with it.

The register strand is divided into *cells*, which are further divided into *domains*. Each domain can be thought of as a fixed-length DNA sequence with relatively weak binding (e.g., 5-7 bases). A strand is stably attached to the register strand only if it is bound by at



■ **Figure 2** Notational conventions and reactions in the SIMD||DNA model. The register strand is on the bottom in each subfigure, with a yellow round “magnetic bead” depicted on the left (not depicted in subsequent figures). Lightly shaded gray or pink regions denote bonds (double-stranded regions), but later figures omit this and simply draw a forward strand (one with 5′ end on left and 3′ end on right) immediately above the domains to which it is bound on the register strand. (a) Conventions for domain names of strands. Domains are numbered $1, \dots, d$ within each cell; $d = 6$ in Figure 2(a) and $d = 18$ in subsequent figures. The register strand has the starred versions of these domains. If a top strand is horizontal over domain i , it has domain i . If it is diagonal over the whole domain, it has an unlabelled domain distinct from all register domains (used as a toehold overhang for detachment, see subfigure (c)). If two strands both partially cover a domain then they both have that domain. (b) A forward instruction strand can attach if at least two complementary consecutive domains are unbound on the register. (c) Reverse instruction strands can bind to toehold overhangs on forward bound strands to detach them from the register. The fact that the (unlabelled) toeholds are complementary is indicated by a diagonal bend in the reverse strand matching. (d) Forward instruction strands can do toehold-mediated strand displacement, one-way if the displacing strand contains all the domains of the displaced strand. (e) If the displacing strand is missing the last domain of the displaced, displacement can also happen, known as *toehold exchange*. This is often called “reversible” since it conserves the number of bound domains, but in the SIMD||DNA model, instruction strands are added in large excess over registers, making it effectively irreversible due to the entropic bias toward binding the instruction strand. Thus it is depicted with irreversible arrows in the figure. (f) Two forward strands can cooperate to displace a single bound top strand, even if neither has enough domains to displace on its own.

least two domains, but one domain is sufficiently long to act as a “toehold” to help initiate strand displacement (Figure 2(c-f)). Within a cell with d domains, each domain is unique and assumed to be named $1, 2, \dots, d$. The register strand has the starred version of these domains, e.g., $1^*, 2^*, 3^*, 4^*, 5^*, 6^*, 1^*, 2^*, 3^*, 4^*, 5^*, 6^*$ reading from the register’s 3′ to 5′ end (left to right) in Figure 2(a). All cells have the same ordered list of domains, so for example in Figure 2(a), domain 5 in cell 1 is the same DNA sequence as domain 5 in cell 2.

An *instruction* is a set of strands that are added to the solution at once. Figure 2(b-f) shows the various reactions that these strands might conduct to change the configuration of strands attached to the register. Multiple reactions can occur in a cascade in a single instruction.¹

¹ See for example instruction 39 in Figure 8. In the right cell, an orange instruction strand displaces an orange strand bound to the register via toehold exchange. This opens a toehold for a blue instruction strand to displace the bound blue strand, resulting in the configuration shown at the beginning of instruction 41.

In particular, the model is nondeterministic, and in general multiple reactions might be possible. It is the job of the system designer to ensure that only one final configuration can result no matter the order of reactions. Instruction strands can either be *forward* (3' arrow on right) or *reverse* (3' arrow on left). Forward instruction strands can do attachment and displacement reactions (Figure 2(b,d-f)) and reverse instruction strands can detach forward strands previously bound to the register (Figure 2(c)).

Crucially, instruction strands are added in large excess over the register strands. Thus even the toehold exchange displacement, which is often considered reversible due to being enthalpically balanced (same number of domains bound before and after), is actually irreversible in the SIMD||DNA model, due to the entropic bias toward binding the new instruction strand with much larger concentration than the strand it displaces.

The notation of [12] uses dashed lines for reverse strands used for detachment, as a visual reminder that they do not bind to the register. We leave reverse strands as solid lines and rely on the 3' arrow to denote that the strand is reversed. We reserve the dashed line notation for later figures to depict *inert* instruction strands: instruction strands that are shown above the register where they would bind if possible, but where no reaction allows them to do so in the current configuration. We also have a slightly different notation for strands with domains mismatching the register: in [12], these are depicted by writing an explicit domain name. In our convention, the drawing of that part of the strand as diagonal and lying entirely above the register domain indicates that the top strand domain and register domain are not complementary (Figure 2(a), cell 2, domain 3). To denote that two adjacent top strands share the same domain, both of which can bind to the register (so they dynamically compete with strand displacement), we draw both strands partially horizontal over the domain, and partially diagonal (Figure 2(a), cell 1, domain 6).

Although these rules allow for nondeterministically competing reactions, our construction is deterministic in the sense that there is only one sequence of reactions possible in any instruction step.

After instruction strands are added and the described reactions go to completion, the *wash* step removes all strands not bound to the register. This includes excess instruction strands that never reacted, as well as strands that were displaced or complexes formed in a detachment reaction.

3 Simulation of Turing machine in SIMD||DNA

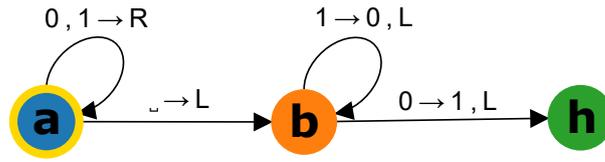
In this section we describe how to simulate an arbitrary 3-symbol single-tape Turing machine with SIMD||DNA instructions.

3.1 High-level overview of construction

Since the SIMD||DNA model as defined has no mechanism to grow the register strand, it can only simulate a fixed-space-bound Turing machine (a.k.a., linear-bounded automaton), which starts with s total tape cells and never moves the tape head off of them. A 3-symbol, space- s Turing machine has three tape symbols: $0, 1, \sqcup$. The binary input $x \in \{0, 1\}^{<s}$ is represented by string $x\sqcup^{s-|x|}$ on the tape in the initial configuration, i.e. x padded with enough blank symbols to make s total tape cells. We use as a running example the 5-transition Turing machine in Figure 3, which increments a binary number.

Each cell of the register represents a tape cell of the Turing machine. If the Turing machine has t total transitions, then each cell uses $d = 2t + 8$ domains.

For each Turing machine, there is a fixed sequence of instructions that, after executing, will update the register to represent the next configuration of the Turing machine.



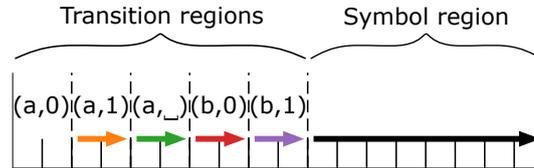
■ **Figure 3** Turing machine (start state a) that increments an integer represented in binary, with the least significant bit on the right. This example is simulated in all subsequent figures.

The cell with the tape head is the only cell with uncovered register domains. Which domains are uncovered (known as a *transition region*) represents both the current state of the Turing machine and the symbol written on that tape cell. For all other cells, a disjoint region (the *symbol region*) represents the symbol on that cell through its pattern of nicks. On the cell with the tape head, the symbol region has no nicks (and represents no symbol) since it is covered by a longer 8-domain strand.

3.2 Representation of Turing machine tape cell as a register cell

In the SIMD||DNA representation of a Turing machine, each register cell represents a single Turing machine tape cell. We represent each Turing machine with tape alphabet $\Gamma = \{0, 1, \sqcup\}$, state set Q , and halt state h , as a set of *transitions*, where each transition $(q, b) \rightarrow (r, c, m)$ means that if the Turing machine is in state $q \in Q \setminus \{h\}$ reading symbol $b \in \Gamma$, it changes to state r , writes symbol c , and moves one cell by $m \in \{L, R\}$ (left or right). Since the Turing machine is deterministic, for each state-symbol pair, there is at most one transition with that pair on the left. (But some such pairs could be undefined, e.g., there is no $(b, \sqcup) \rightarrow \dots$ transition in Figure 3.)

Representation of tape cell with tape head



■ **Figure 4** A SIMD||DNA cell where the tape head is presently located. The $(a, 0)$ region is fully exposed, indicating that the Turing machine is in state a and that the cell contains the symbol 0. The other transition regions are fully covered, and the symbol region (rightmost 8 domains of the cell) is covered by a single long strand, not encoding any symbol (which is encoded by the uncovered transition region).

We take every state-symbol pair $(q, \sigma) \in (Q \setminus \{h\}) \times \Gamma$ (each possible left side of a transition) and represent each as two consecutive domains in a SIMD||DNA register cell. See Figure 4. Recall the binary incrementing Turing machine of Figure 3. It has five transitions: $(a, 0) \rightarrow (a, 0, R)$, $(a, 1) \rightarrow (a, 1, R)$, $(a, \sqcup) \rightarrow (b, \sqcup, L)$, $(b, 0) \rightarrow (1, h, L)$, $(b, 1) \rightarrow (0, b, L)$. We call the pair on the left the *transition input*. Each of the given transition inputs is represented in the SIMD||DNA cell using two domains, requiring ten domains total for our example. Since each register cell represents a cell in M , we must denote the presence of the tape head on one of the cells. If the tape head is present on a given cell and if the current Turing machine configuration has a valid transition, then the two domains that represent

14:6 Simulating 3-Symbol Turing Machines with SIMD||DNA

that transition will have no top strand attached to them, leaving them exposed. For example, if the tape head is on a cell with the 0 symbol, and the Turing machine is currently in state a , then the region that represents $(a, 0)$ in that cell will be exposed to serve as a toehold for strand displacement. The other transition regions are fully covered by 2-domain strands.

Representation of tape cell without tape head

If the tape head is not present on a cell, or if no valid transitions exist for the current configuration,² then every transition region is covered by 2-domain strands. Eight additional domains at the rightmost part of the cell, called the **symbol region** represent the current symbol written on that cell.

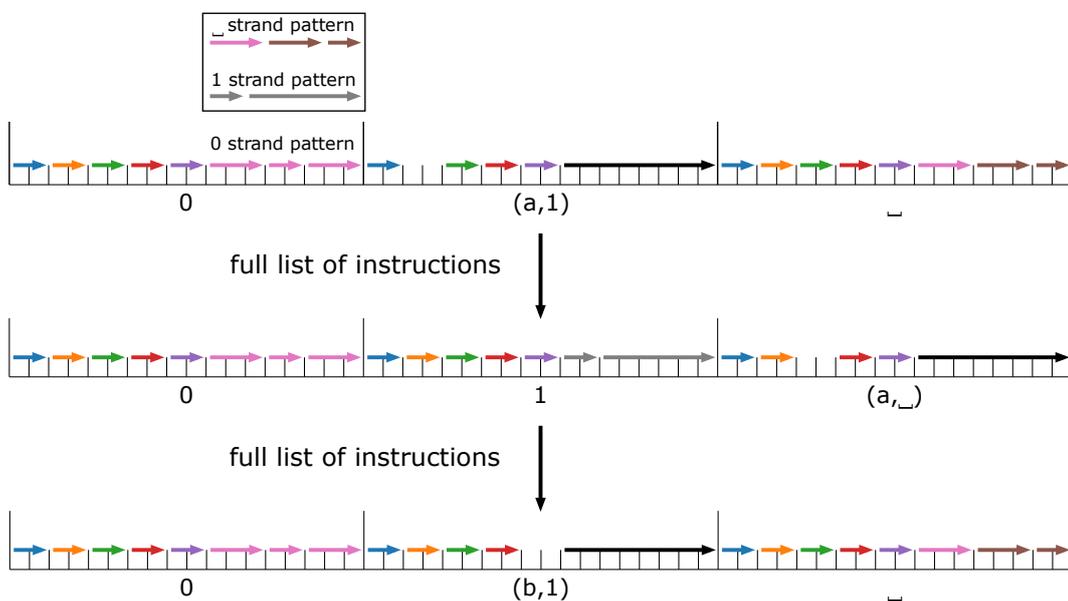
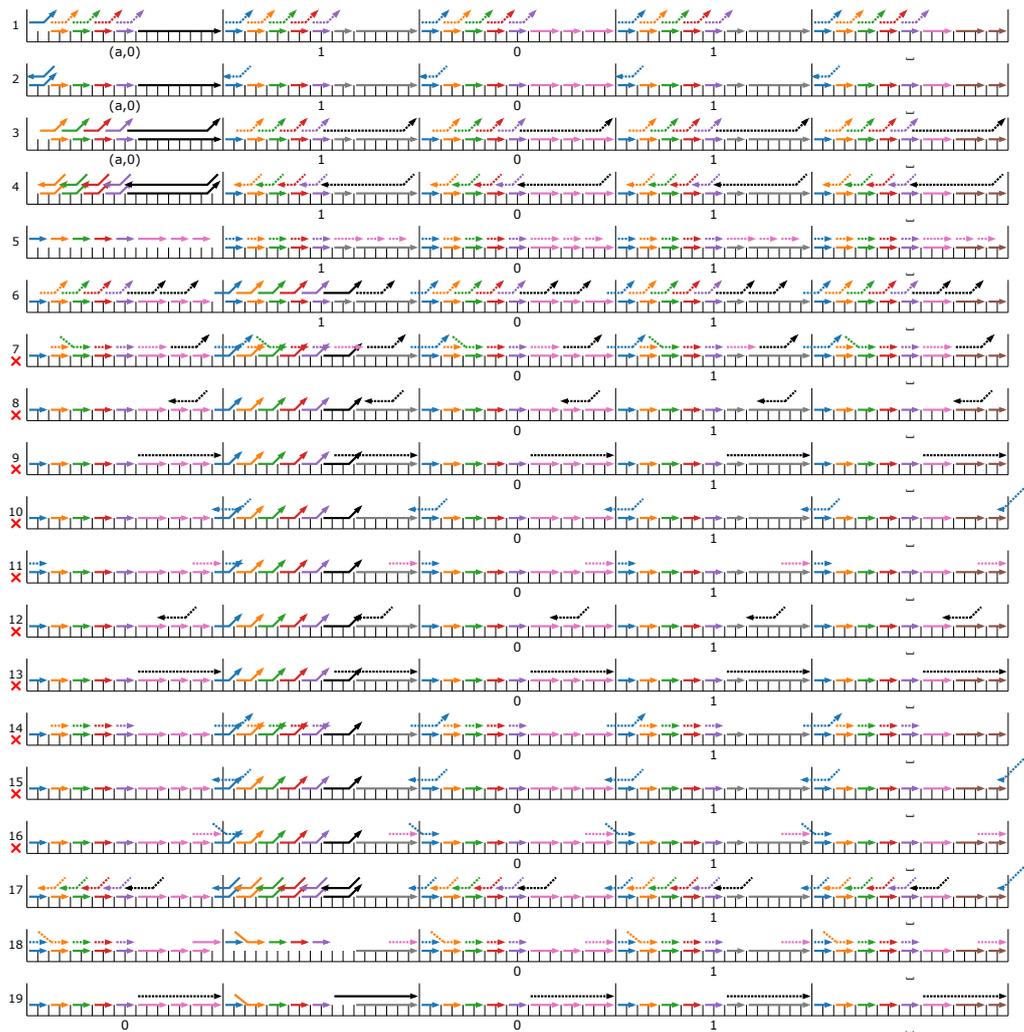


Figure 5 High-level overview of construction. A Turing machine register currently in state a , with the tape head on the second cell. The second cell contains the symbol 1. The leftmost cell contains symbol 0. The inset above shows encodings for 1 and \sqcup . All the transition regions are fully covered in cells lacking the tape head. After the full list of instructions in the SIMD||DNA program are complete, the register represents the Turing machine configuration with state a and the tape head moved to the rightmost cell with the \sqcup . The same full list of instructions updates the register again, now representing the Turing machine configuration in state b with the tape head back on the middle cell.

Whenever the tape head is present on a cell, the symbol region is covered by a single 8-domain strand that does not encode any symbol, since the symbol information is already encoded in the transition region with an open toehold.

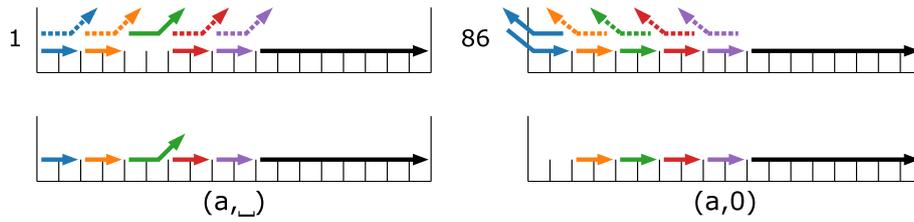
² For example, if the machine has halted; see the bottom register configuration of Figure 9 for a case where the state is non-halting but no valid transition exists.



■ **Figure 6** An overview of the first 19 instructions of the construction, which represent the $(a, 0) \rightarrow (a, 0, R)$ transition of the Turing machine shown in Figure 3. Instructions marked with a red cross are fully inert, meant for cases not exhibited by this register.

3.3 Detailed description of SIMD||DNA instructions simulating a Turing machine

We designed an algorithm that converts Turing machine specifications from <https://turingmachine.io> into SIMD||DNA representations, along with the equivalent instructions. Each transition τ_i has an associated sublist of instructions L_i , and, not knowing which transition is applicable to the current configuration, we simply add instruction strands in order from L_1, L_2, \dots . For $i \neq j$, to ensure that L_j instructions have no effect when the current applicable transition is τ_i , we “plug” the open domains of other transition regions with a strand and remove the plug strand once it’s time to process that transition. Because the SIMD||DNA model allows parallel computation among multiple registers in the same solution, this prevents instructions meant for one configuration from affecting registers currently not in that configuration. In the beginning, all transition regions are plugged, where the order of processing for the transitions is arbitrary.



■ **Figure 7** On the left, the first instruction in the whole SIMD||DNA program covers the transition region representing the next applicable transition. The two horizontal rows of strands have the following interpretation: Bottom are strands bound to register, top are instruction strands. Dashed instruction strands will not have an effect on the current cell (but to help verify correctness, they are shown above where they would bind to the register). On the right, the last instruction in the whole SIMD||DNA program, which removes the post-plug strands in each register. In the above example, the post-plug strand covers the $(a, 0)$ transition region, indicating the cell's next Turing machine transition. After this, the entire register is updated to appear as a configuration similar to those in Figure 5.

Pre-plug and post-plug strands to protect instructions for inapplicable transitions from affecting configuration

The full list of instructions to simulate a Turing machine transition works as follows. Recall that in the “clean” configurations shown in Figure 5, the only exposed register domains are on the cell representing the tape head. The first instruction in the entire list contains *pre-plug* strands for each transition region. At the end of each instruction sublist L_i , a *post-plug strand* is also placed on the transition region that represents the Turing machine's next applicable transition. Examples of both can be seen in Figure 7. These strands act like a “chemical protecting group” that prevents instruction sublists L_i from modifying the register unless they apply to the intended transition. The difference between the pre-plug and post-plug strands is that pre-plug strands protect the configuration when using instructions strands *before* the applicable transition, whereas post-plug strands protect the configuration when using instructions strands *after* the applicable transition. In the left part of Figure 7, because the register has a pre-plug strand in (a, \sqcup) , it means that the instruction sublist $L_{(a, \sqcup)}$ has not been applied to it yet. Instruction sublists for the other transitions $(a, 0) \rightarrow \dots$, $(a, 1) \rightarrow \dots$, $(b, 0) \rightarrow \dots$, $(b, 1) \rightarrow \dots$ will be inert, not affecting the register. The first instruction of $L_{(a, \sqcup)}$ will remove this pre-plug strand so that any register in the (a, \sqcup) configuration can be processed. The instruction sublists will result in a configuration like that of the bottom of Figures 8 and 9. Figure 9 shows instructions that affect the cell where the tape head *was* (right cell), not where it will be *next* (left cell), which is why the left cell is the same in both Figures 8 and 9. This almost represents the next Turing machine configuration, but with the appropriate transition region covered by a *post-plug strand*. The final instruction in the entire list (Figure 7) removes this post-plug strand, restoring the register configuration to be as shown in Figure 5.

The post-plug strand placed on the transition region at the end of simulating a transition has a different purpose from the pre-plug strand placed in instruction 1. Its purpose is to prevent the register from updating its state multiple times in the same instruction iteration. For example, if a register has a post-plug strand on the $(b, 1)$ region (indicating that it has been processed and that its next transition is $(b, 1)$), and the instruction sublist that processes $(b, 1)$ comes after, the register will be unaffected by $(b, 1)$'s deprotecting instruction, keeping it inert throughout. The final instruction in the entire iteration removes these post-plug strands from the registers, as seen in Figure 7, preparing the registers for the next iteration of the instruction set.

Note the duality between pre-plug and post-plug instructions. *All* pre-plug strands are included in the first instruction, though only one of them will bind (the one matching the applicable transition), and instruction strands removing *all* post-plug strands are included as part of the last instruction, though only one will find its complementary post-plug strand to remove. On the other hand, each pre-plug instruction is removed more specifically, by adding a single complementary strand to remove it just prior to the sublist of instructions corresponding to the applicable transition. Similarly, each post-plug strand is added by itself, at the end of the instruction sublist corresponding to the applicable transition.

In the next section, we will describe the details of the instruction sublists that represent the Turing machine transitions.

Sublist of instructions representing a single Turing machine transition

Figure conventions. For the figures explaining SIMD||DNA instructions that simulate a single transition of the Turing machine (Figure 8 and beyond), we use the following conventions in figures. Several register configurations are shown, but they are not necessarily consecutive. Each is numbered with its absolute index in the list of all 86 instructions implementing the Turing machine of Figure 3. If two adjacent configurations have non-consecutive instruction indices, this means that the instructions not shown are inert: their strands do not affect the register in that configuration. The instruction strands that have just been added are always shown above the register, with a solid line if they will do a reaction as in Figure 2, and with a dashed line if that instruction strand is inert for that configuration. The final configuration in each figure does not show any instruction strands, but for all figures there is a followup figure showing what happens next from that configuration (possibly the followup is Figure 7, the final instruction in the entire program, removing the post-plug strand from the next applicable transition region).

Each Turing machine transition is individually processed by a sublist of instructions. The pre-plug strand is first removed by an instruction containing its complementary strand, so that its corresponding transition region in the cell can be used as a toehold, such as instruction 38 in Figure 8. The next instructions then update the contents of the current cell to encode the symbol that the tape head writes. For example, in Figure 9, the transition $(a, \sqcup) \rightarrow (b, \sqcup, L)$ is represented, and the strand encoding of \sqcup is placed in the right cell after the tape head writes on it and moves left. After that, the instructions check the contents of the tape head's new location and determine the Turing machine's next configuration. In Figure 8, the tape head moves to the left cell and finds a 1, and the Turing machine goes to state b , so it leaves a post-plug strand on $(b, 1)$'s transition region to show that the register has been processed for that instruction iteration, as seen in instruction 46.

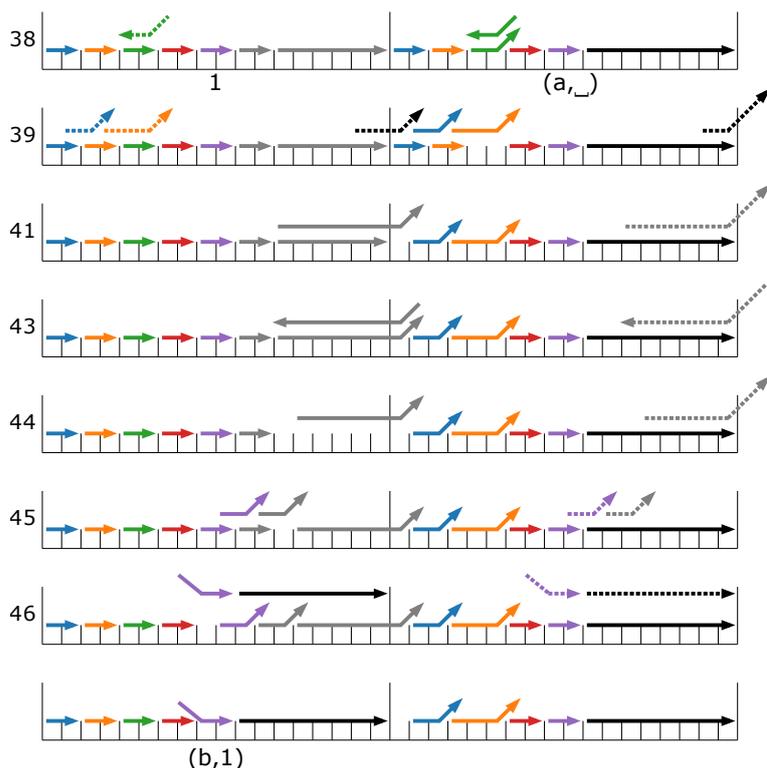
Left versus right tape head moves

In the SIMD||DNA instructions implementing a single transition, there are two sublists: next-cell instructions and previous-cell instructions. As their names indicate, next-cell instructions update the contents of the tape head's destination, while previous-cell instructions update the contents of the tape head's former location. Other factors such as the symbol to be written on the current cell and the next applicable transition region only introduce minor variations in the instruction strands.

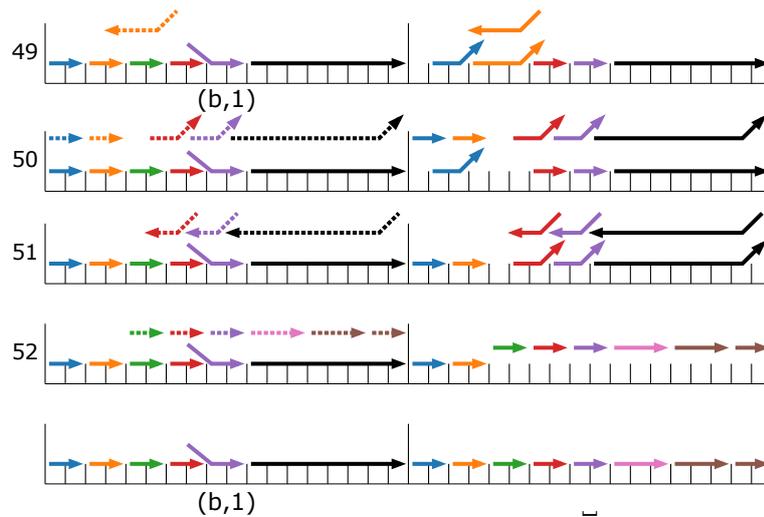
For transitions moving the tape head *left*, the next-cell instructions precede the previous-cell instructions (see Figures 8–11). For transitions moving the tape head *right*, this order is reversed (see Figure 12 and Appendix A in the full version of the paper.).

Left tape head moves

Figure 8 shows the next-cell instructions for transition $(a, \sqcup) \rightarrow (b, \sqcup, L)$, for the special case when the cell to the left of the tape head has the symbol 1. Figure 10 shows the next-cell instructions for the same transition when the cell to the left of the tape head has the symbol 0, and Figure 11 shows the next-cell instructions when the cell to the left has the symbol \sqcup . Note that in any given configuration, the same instructions will result in exactly one of the situations depicted in Figures 8, 10, and 11. Once these next-cell instructions are applied, the leftmost domain of the previous cell will serve as a toehold for the previous-cell instructions that follow in Figure 9.



■ **Figure 8** First half of instructions (next-cell instructions) to implement transition $a, \sqcup \rightarrow b, \sqcup, L$, in the case that the cell to the left (where the tape head will move) has a symbol 1. After instruction 46, the left cell (where the tape head will move next) now encodes the next state b and the symbol 1 on the new tape cell. Rather than show a red cross, inert instructions are instead omitted in this figure and the following ones. Inert instructions (40,42,47,48) are used when the cell to the left has a symbol 0 or \sqcup on it instead. Figures 10 and 11 respectively show these cases. Figure 9 shows instructions completing the transition by writing \sqcup over the right cell. In instruction 38, the transition region is first unplugged to expose the toehold. The strands in instruction 39 cascade until the left cell's symbol region, allowing the instructions to branch out depending on the tape content of the left cell. The remaining instructions process the left cell so that it encodes the next configuration. In instruction 46, a special post-plug strand whose leftmost domain is orthogonal is attached to the region that represents the next configuration; this strand will be removed once all transitions have been processed. The angled dashed strands to the right of the register indicate that no cells are present to the right of the rightmost cell; if there were another cell, then one more domain of each of these strands would be horizontal, bound to the leftmost domain of the cell to the right (just as with their solid counterparts to the left).



■ **Figure 9** Second half (previous-cell instructions) of transition $a, \sqcup \rightarrow b, \sqcup, L$ whose first 11 instructions are shown in Figure 8, Figure 10, and Figure 11; these instructions write \sqcup over the old cell (right of Figure 8) where the tape head was at the start of the transition. Slight variations of instruction 15 write 0 or 1 instead of \sqcup .

Right tape head moves

For right transitions, the first three instructions are the previous-cell instructions, as shown in Figure 12. The next-cell instructions follow, where the instructions strands that apply to the register depend on the contents of the cell to the right of the tape head (a 0, 1, or \sqcup .) The figures depicting these are shown in Appendix A of the full version of the paper.

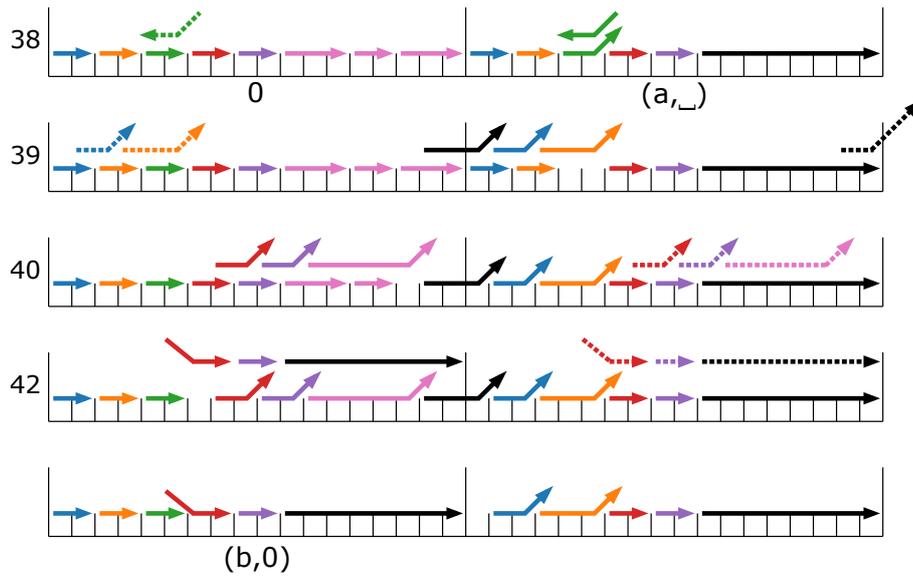
Final deprotecting instruction

Figure 7 shows the final deprotecting instruction, which removes the post-plug strand put in place during the last instruction of the non-inert instruction sublists. This puts the register back into a “clean” configuration representing a Turing machine configuration, such as those shown in Figure 5, opening up new toeholds for the next iteration of instructions.

3.4 Complexity of construction

A common metric of “complexity” of DNA systems is the number of unique domains they require. Fewer is better because it is a nontrivial task to design *orthogonal* domains: domains that, if they are not perfectly complementary, will have low binding affinity. Low domain complexity is particular important in SIMD||DNA, where each domain is considered “toehold-length”: sufficiently short (5-7 bases) that the off-rate of a strand bound by a single domain is large enough to detach in a short amount of time. There are only $4^7 = 16384$ DNA sequences of length 7. In practice even fewer are available: half are complementary to the other half, leaving only 8192 available to assign to the unstarred versions of each domain. DNA sequence design heuristics such as the “3-letter code” (using only A,T,C for forward strands, thus only A,T,G for the register and reverse strands) reduce this number to $3^7 = 2187$. Reasonable design constraints, e.g., avoiding almost equal domains such as $5'-AAAAAAG-3'$ and $5'-AAAAAAA-3'$, which both bind almost equally strongly to $3'-TTTTTTT-5'$, further limit the set of available domains.

14:12 Simulating 3-Symbol Turing Machines with SIMD||DNA



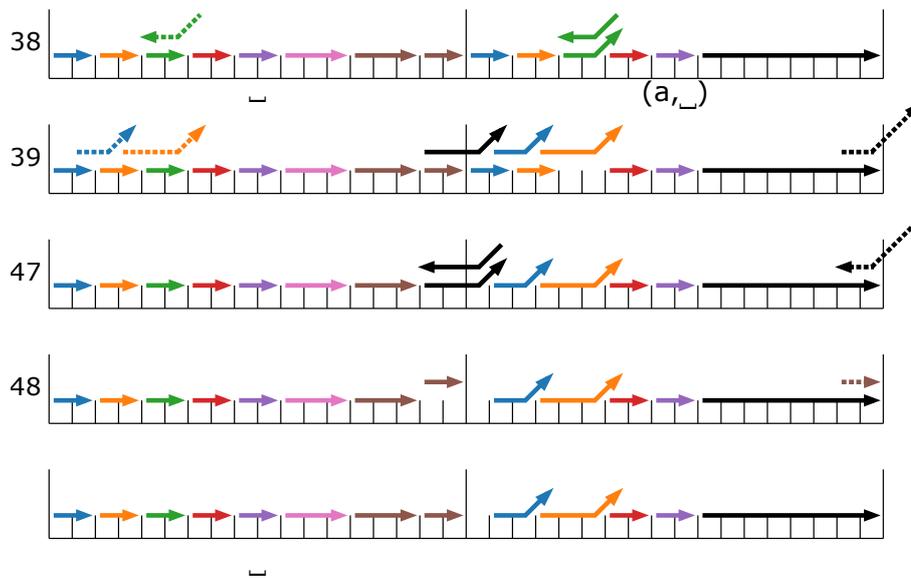
■ **Figure 10** The next-cell instructions of the transition in Figure 8 that are applicable when the cell to the left of the tape head has a 0.

Our construction uses $d = 2t + 8$ total unique DNA domains (which repeat throughout the register), where t is the number of transitions of the simulated Turing machine. Each transition is represented by 2 domains, assumed to be bound strongly enough not to spontaneously dissociate. (The construction can be altered to use more domains, in case this assumption is overly ideal.) To simulate a Turing machine with space bound s , the register has s “cells”, where each cell is simply a copy of one each of the d domains $1, \dots, d$. Thus, if each domain consists of k nucleotides, the register strand has $k \cdot s \cdot (2t + 8)$ total nucleotides. There is an 11-state, 3-symbol universal Turing machine (directly simulating another Turing machine) with 32 transitions [6], giving $2 \cdot 32 + 8 = 72$ total domains required in the worst case. However, specialized non-universal Turing machines with a smaller number of transitions (for example the 5-transition binary incrementor of Figure 5) could accomplish many computationally sophisticated tasks.

Each Turing machine transition can be represented by approximately 16 SIMD||DNA instructions. The exact number varies depending on the specific properties of the transition in question, such as the the direction of the tape head, the symbol to be written, and whether any of the possible next configurations are halting or not. This range has constant upper and lower bounds, however, so the total number of instructions is in $O(t)$, where t is the number of transitions in the Turing machine. Because the size of the cell increases in $O(t)$ due to the transition regions, the number of DNA strands present in some instructions also scales by a factor of $O(t)$, most notably the instructions that cause a cascade of toehold exchanges.

4 Conclusion

Our construction, like the Rule 110 simulation of Wang, Chalk, and Soloveichik [12], is not Turing universal because it simulates a *space-bounded* Turing machine. Truly universal computation should be possible without advanced knowledge of the space requirement. An interesting question (raised also in [12]) is whether a suitable augmentation of the SIMD||DNA model could allow Turing-universal computation. This would require unbounded polymers

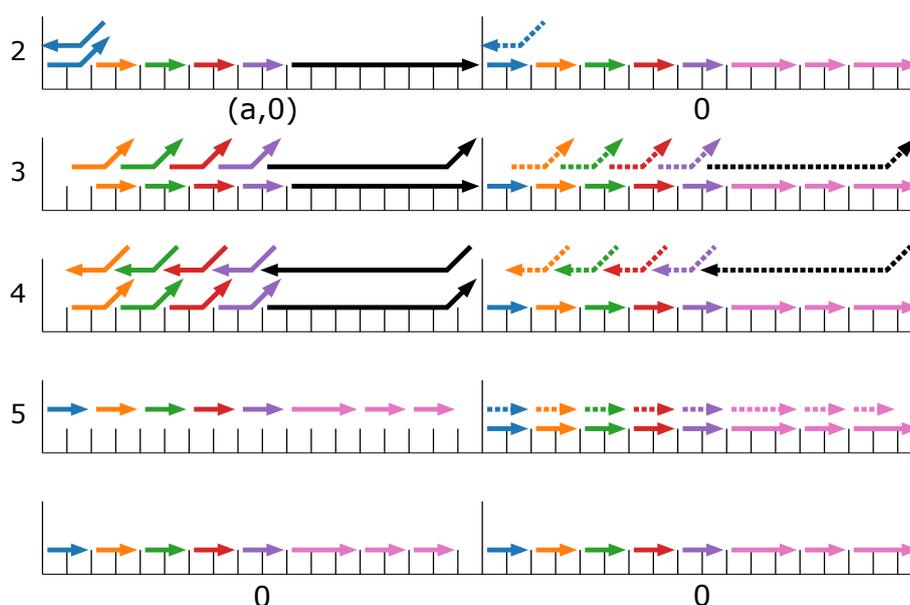


■ **Figure 11** The next-cell instructions of the transition in Figure 8 that are applicable when the cell to the left of the tape head has a \sqcup . Because no transition exists for the state-symbol pair (b, \sqcup) , the left cell is left unchanged.

such as those used in the two-stack machine DNA implementation of Qian, Soloveichik, and Winfree [8]. That paper showed Turing universal computation in the case where only a single copy of certain strands are permitted to exist in solution, simulating only a single stack machine at a time, in contrast to the SIMD||DNA model, where we can operate on many registers, each representing their own Turing machine, in parallel.

Technically the strand displacement reactions of the SIMD||DNA model are currently powerful enough to grow arbitrarily large polymers from a fixed set of strands, as in [8], by alternating top and bottom strands, making a long double-helix with nicks on the top *and* bottom. However, it is difficult to see how to use the ability of SIMD||DNA to exploit this to simulate a Turing machine represented in this way. If there are multiple bottom strands, i.e., there is a nick on the bottom, then any strand displacement of top strands, upon reaching this nick, would separate the polymer into two complexes to the left and right of this nick, and the right polymer would be lost in the wash step. One could imagine, however, augmenting the model to allow, for example, 3-arm junctions, which could be used to do strand displacement that crosses over the boundary between two bottom strands without separating them (since they would be joined to each other by a strong domain representing the third arm “below” the main helix).

Although some of the toehold exchanges in the SIMD||DNA model are reversible based on the principles of DNA strand displacement, we make the assumption that the applied instructions are not undone by the displaced strands. This is based on the assumption that the instruction strands are present in a sufficiently high concentration that reversal is unlikely. Because multiple registers can be present in the same solution, another possibility to consider is a displaced DNA strand from Register A binding to an open toehold in Register B, such that the attachment is irreversible even with the presence of a high concentration of instruction strands. One open question is to design a system that factors in these possibilities, reducing the likelihood of unexpected strand displacement results.



■ **Figure 12** First half (previous-cell instructions) of transition $a,0 \rightarrow a,0,R$. Instruction 3's strands cascade to the right side of the current cell, while instruction 4 removes the previously introduced strands. Instruction 5 then covers up all the transition regions and adds the strands of the symbol to be written on the current cell (0 in this example), but leaving the rightmost domain exposed to act as a toehold for the next instructions.

Another open question is whether a more domain-efficient encoding exists for the Turing machine construction. Given n transitions, $2n$ domains are required to represent them, which has $O(n)$ complexity. However, given d domains, there are 2^{d-1} possible nick patterns among the attached strands, which makes $O(\log n)$ domain complexity possible in theory.

References

- 1 James Bornholt, Randolph Lopez, Douglas M Carmean, Luis Ceze, Georg Seelig, and Karin Strauss. A DNA-based archival storage system. In *ASPLOS 2016: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 637–649, 2016.
- 2 Tonglin Chen, Arnav Solanki, and Marc Riedel. Parallel Pairwise Operations on Data Stored in DNA: Sorting, Shifting, and Searching. In *DNA 27: 27th International Conference on DNA Computing and Molecular Programming*, volume 205, pages 11:1–11:21, 2021. doi: 10.4230/LIPIcs.DNA.27.11.
- 3 George M Church, Yuan Gao, and Sriram Kosuri. Next-generation digital information storage in DNA. *Science*, 337(6102):1628–1628, 2012.
- 4 Matthew Cook. Universality in elementary cellular automata. *Complex systems*, 15(1):1–40, 2004.
- 5 Turlough Neary and Damien Woods. P-completeness of cellular automaton Rule 110. In *ICALP 2006: International Colloquium on Automata, Languages, and Programming*, pages 132–143. Springer, 2006.
- 6 Turlough Neary and Damien Woods. Small fast universal Turing machines. *Theoretical Computer Science*, 362(1-3):171–195, 2006.

- 7 Lee Organick, Siena Dumas Ang, Yuan-Jyue Chen, Randolph Lopez, Sergey Yekhanin, Konstantin Makarychev, Miklos Z Racz, Govinda Kamath, Parikshit Gopalan, Bichlien Nguyen, Christopher N Takahashi, Sharon Newman, Hsing-Yeh Parker, Cyrus Rashtchian, Kendall Stewart, Gagan Gupta, Robert Carlson, John Mulligan, Douglas Carmean, Georg Seelig, Luis Ceze, and Karin Strauss. Random access in large-scale DNA data storage. *Nature Biotechnology*, 36(3):242, 2018.
- 8 Lulu Qian, David Soloveichik, and Erik Winfree. Efficient Turing-universal computation with DNA polymers. In *International Workshop on DNA-Based Computers*, pages 123–140. Springer, 2010.
- 9 Georg Seelig, David Soloveichik, David Yu Zhang, and Erik Winfree. Enzyme-free nucleic acid logic circuits. *Science*, 314(5805):1585–1588, 2006.
- 10 SIMD||DNA simulator.
Source code: <https://github.com/UC-Davis-molecular-computing/simd-dna>, 2021.
- 11 S Kasra Tabatabaei, Boya Wang, Nagendra Bala Murali Athreya, Behnam Enghiad, Alvaro Gonzalo Hernandez, Christopher J Fields, Jean-Pierre Leburton, David Soloveichik, Huimin Zhao, and Olgica Milenkovic. DNA punch cards for storing data on native DNA sequences via enzymatic nicking. *Nature Communications*, 11(1):1–10, 2020.
- 12 Boya Wang, Cameron Chalk, and David Soloveichik. SIMD||DNA: Single instruction, multiple data computation with DNA strand displacement cascades. In *DNA 2019: International Conference on DNA Computing and Molecular Programming*, pages 219–235, 2019.

Parameterized Temporal Exploration Problems

Thomas Erlebach  

Department of Computer Science, Durham University, UK

Jakob T. Spooner  

School of Computing and Mathematical Sciences, University of Leicester, UK

Abstract

In this paper we study the fixed-parameter tractability of the problem of deciding whether a given temporal graph \mathcal{G} admits a temporal walk that visits all vertices (temporal exploration) or, in some problem variants, a certain subset of the vertices. Formally, a temporal graph is a sequence $\mathcal{G} = \langle G_1, \dots, G_L \rangle$ of graphs with $V(G_t) = V(G)$ and $E(G_t) \subseteq E(G)$ for all $t \in [L]$ and some underlying graph G , and a temporal walk is a time-respecting sequence of edge-traversals. For the strict variant, in which edges must be traversed in strictly increasing timesteps, we give FPT algorithms for the problem of finding a temporal walk that visits a given set X of vertices, parameterized by $|X|$, and for the problem of finding a temporal walk that visits at least k distinct vertices in V , parameterized by k . For the non-strict variant, in which an arbitrary number of edges can be traversed in each timestep, we parameterize by the lifetime L of the input graph and provide an FPT algorithm for the temporal exploration problem. We also give additional FPT or $W[2]$ -hardness results for further problem variants.

2012 ACM Subject Classification Theory of computation \rightarrow Graph algorithms analysis; Theory of computation \rightarrow Fixed parameter tractability

Keywords and phrases Temporal graphs, fixed-parameter tractability, parameterized complexity

Digital Object Identifier 10.4230/LIPIcs.SAND.2022.15

Funding *Thomas Erlebach*: Supported by EPSRC grants EP/S033483/2 and EP/T01461X/1.

1 Introduction

The problem of computing a series of consecutive edge-traversals in a static (i.e., classical discrete) graph G , such that each vertex of G is an endpoint of at least one traversed edge, is a fundamental problem in algorithmic graph theory, and an early formulation was provided by Shannon [26]. Such a sequence of edge-traversals might be referred to as an *exploration* or *search* of G and, from a computational standpoint, it is easy to check whether a given graph G admits such an exploration and easy to compute one if the answer is yes – we simply carry out a depth-first search starting at an arbitrary start vertex in $V(G)$ and check whether every vertex of G is reached. We consider in this paper a decidedly more complex variant of the problem, in which we try to find an exploration of a *temporal graph*. A temporal graph $\mathcal{G} = \langle G_1, \dots, G_L \rangle$ is a sequence of static graphs G_t such that $V(G_t) = V(G)$ and $E(G_t) \subseteq E(G)$ for any *timestep* $t \in [L]$ and some fixed *underlying graph* G .

A concerted effort to tackle algorithmic problems defined for temporal graphs has been made in recent years. With the addition of time to a graph’s structure comes more freedom when defining a problem. Hence, many studies have focused on temporal variants of classical graph problems; for example, the travelling salesperson problem [21]; shortest paths [27]; vertex cover [3]; maximum matching [20]; network flow problems [1]; and a number of others. For more examples, we point the reader to the works of Molter [23] or Michail [21]. One seemingly common trait of the problems that many of these studies consider is the following: *Problems that are easy for static graphs often become hard on temporal graphs, and hard problems for static graphs remain hard on temporal graphs.* This certainly holds true for the problem of deciding whether a given temporal graph \mathcal{G} admits a *temporal walk* W – roughly



© Thomas Erlebach and Jakob T. Spooner;

licensed under Creative Commons License CC-BY 4.0

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).

Editors: James Aspnes and Othon Michail; Article No. 15; pp. 15:1–15:17

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

speaking, a sequence of edges traversed consecutively and during strictly increasing timesteps – such that every vertex of \mathcal{G} is an endpoint of at least one edge of W (any temporal walk with this property is known as an *exploration schedule*). Indeed, Michail and Spirakis [22] showed that this problem, TEMPORAL EXPLORATION or TEXP for short, is NP-complete. In this paper, we consider variants of the TEXP problem from a fixed-parameter perspective and under both *strict* and *non-strict* settings. More specifically, we consider problem variants in which we look for *strict* temporal walks that traverse each consecutive edge at a timestep strictly larger than the previous, as well as variants that ask for *non-strict* temporal walks that allow an unlimited but finite number of edges to be traversed in each timestep.

Contribution. In Section 2 we prove FPT-membership for two natural parameterized variants of TEXP. Firstly, we parameterize by the size k of a fixed subset of the vertex set and ask for an exploration schedule that visits at least these vertices, providing a $O(2^k k Ln^2)$ -time algorithm. Secondly, we parameterize by only an integer k and ask that a computed solution visits at least k arbitrary vertices – in this case we specify, for any $\varepsilon > 0$, a randomized algorithm (based on the colour-coding technique first introduced by Alon et al. [4]) with running time $O((2e)^k Ln^3 \log \frac{1}{\varepsilon})$. A now-standard derandomization technique due to Naor et al. [24] is then utilized in order to obtain a deterministic $(2e)^k k^{O(\log k)} Ln^3 \log n$ -time algorithm.

In Section 3, we consider the non-strict variant known as NON-STRICT TEMPORAL EXPLORATION, or NS-TEXP, which was introduced in [17]. Here, a candidate exploration schedule is permitted to traverse an unlimited but finite number of edges during each timestep, and it is not too hard to see that this change alters the problem’s structure quite drastically (more details in Section 3). We therefore use a different model of temporal graphs to the one considered in Section 2, which we properly define later. For this problem, we parameterize by the length L of the sequence of static graphs that comprises our input temporal graph, and provide an $O(L(L!)^2 n)$ -time recursive search-tree algorithm. We also consider a generalized variant, SET NS-TEXP, in which we are supplied with m subsets of the input temporal graph’s vertex set and are asked to decide whether there exists a non-strict temporal walk that visits at least one vertex belonging to each set; this problem is shown to be $W[2]$ -hard via a reduction from SET COVER.

Related work. We refer the interested reader to Casteigts et al. [11] for a study of various models of dynamic graphs, and to Michail [21] for an introduction to temporal graphs and some of their associated combinatorial problems. Brodén et al. [8] consider the TEMPORAL TRAVELLING SALESPERSON PROBLEM for complete temporal graphs with n vertices. The costs of edges are allowed to differ between 1 and 2 in each timestep. They show that when an edge’s cost changes at most k times during the input graph’s lifetime, the problem is NP-complete, but provide a $(2 - \frac{2}{3k})$ -approximation. For the same problem, Michail and Spirakis [22] prove APX-hardness and provide a $(1.7 + \epsilon)$ -approximation. Bui-Xuan et al. [9] propose multiple objectives for optimisation when computing temporal walks/paths: e.g., *fastest* (fewest number of timesteps used) and *foremost* (arriving at the destination at the earliest time possible). Michail and Spirakis [22] introduced the TEXP problem, which asks whether or not a given temporal graph admits a temporal walk that visits all vertices at least once. The problem is shown to be NP-complete when no restrictions are placed on the input, and they propose considering the problem under the *always-connected* assumption as a means of ensuring that exploration is possible (provided the lifetime of the input graph is sufficiently long). Erlebach et al. [16] consider the problem of computing foremost exploration schedules

under the always-connected assumption, proving $O(n^{1-\varepsilon})$ -inapproximability (for any $\varepsilon > 0$) amongst other results. Bodlaender and van der Zanden [6] examined the TEXP problem when restricted to temporal graphs whose underlying graph has pathwidth at most 2 and that are connected in each timestep, showing the problem to be NP-complete in this case. Akrida et al. [2] consider a TEXP variant called RETURN-TO-BASE TEXP, in which the underlying graph is a star and a candidate solution must return to the vertex from which it initially departed (the star's centre). They prove various hardness results and provide polynomial-time algorithms for some special cases. Casteigts et al. [12] studied the fixed-parameter tractability of the problem of finding temporal paths between a source and destination that wait no longer than Δ consecutive timesteps at any intermediate vertex. Bumpus and Meeks [10] considered, again from a fixed-parameter perspective, a temporal graph exploration variant in which the goal is no longer to visit all of the input graph's vertices at least once, but to traverse all edges of its underlying graph exactly once (i.e., computing a temporal Eulerian circuit). The problem of NON-STRICT TEMPORAL EXPLORATION was introduced and studied in [17]. Here, a computed walk may make an unlimited number of edge-traversals in each given timestep. Amongst other things, NP-completeness of the general problem is shown, as well as $O(n^{1/2-\varepsilon})$ and $O(n^{1-\varepsilon})$ -inapproximability for the problem of minimizing the arrival time of a temporal exploration in the cases where the number of timesteps required to reach any vertex v from any vertex u is bounded by $c = 2$ and $c = 3$, respectively. Notions of strict/non-strict paths which respectively allow for a single edge/unlimited number of edge(s) to be crossed in any timestep have been considered before, notably by Kempe et al. [19] and Zschoche et al. [28].

Preliminaries. For a pair of integers x, y with $x \leq y$ we denote by $[x, y]$ the set $\{z : x \leq z \leq y\}$; if $x = 1$ we write $[y]$ instead. We use standard terminology from graph theory [14], and we assume any static graph $G = (V, E)$ to be simple and undirected. A parameterized problem is a language $L \subseteq \Sigma^* \times \mathbb{N}$, where Σ is a finite alphabet. For an instance $(I, k) \in \Sigma^* \times \mathbb{N}$, k is called the parameter. The problem is in FPT (fixed-parameter tractable) if there is an algorithm that solves every instance in time $f(k) \times |I|^{O(1)}$ for some computable function f . A proof that a problem is hard for complexity class $W[r]$ for some integer $r \geq 1$ is seen as evidence that the problem is unlikely to be contained in FPT. For more on parameterized complexity, including definitions of the complexity classes $W[r]$, we refer to [15, 13]. We defer formal definitions of both the strict and non-strict variants of TEXP, as well as their associated temporal graph models, to Sections 2 and 3 respectively.

2 Strict TEXP parameterizations

We begin with the definition of a temporal graph:

► **Definition 1** (Temporal graph). *A temporal graph \mathcal{G} with underlying graph $G = (V, E)$, lifetime L and order n is a sequence of simple undirected graphs $\mathcal{G} = \langle G_1, G_2, \dots, G_L \rangle$ such that $|V| = n$ and $G_t = (V, E_t)$ (where $E_t \subseteq E$) for all $t \in [L]$.*

For a temporal graph $\mathcal{G} = \langle G_1, \dots, G_L \rangle$, the subscripts $t \in [L]$ indexing the graphs in the sequence are referred to as *timesteps* (or *steps*) and we call G_t the t -th layer. A tuple (e, t) with $e \in E(G)$ is an *edge-time pair* (or *time edge*) of \mathcal{G} if $e \in E_t$. Note that the size of any temporal graph (i.e., the maximum number of time edges) is bounded by $O(Ln^2)$.

► **Definition 2** (Strict temporal walk). *A strict temporal walk W in \mathcal{G} is a tuple $W = (t_0, S)$, consisting of a start time t_0 and an alternating sequence of vertices and edge-time pairs $S = \langle v_1, (e_1, t_1), v_2, (e_2, t_2), \dots, v_{l-1}, (e_{l-1}, t_{l-1}), v_l \rangle$ such that $e_i = \{v_i, v_{i+1}\}$, $e_i \in G_{t_i}$ for $i \in [l-1]$ and $1 \leq t_0 \leq t_1 < t_2 < \dots < t_{l-1} \leq L$.*

15:4 Parameterized Temporal Exploration Problems

We say that a temporal walk $W = (t_0, S)$ *visits* any vertex that is included in S . Further, W *traverses* edge e_i at time t_i for all $i \in [l - 1]$ and is said to *depart from* (or *start at*) $v_1 \in V(\mathcal{G})$ at timestep t_0 and *arrive at* (or *finish at*) $v_l \in V(\mathcal{G})$ at the end of timestep t_{l-1} (or, equivalently, at the beginning of timestep $t_{l-1} + 1$). Its *arrival time* is defined to be $t_{l-1} + 1$. It is assumed that W is positioned at v_1 at the start of timestep $t_0 \in [t_1]$ and waits at v_1 until edge e_1 is traversed during timestep t_1 . The quantity $|W| = t_{l-1} - t_0 + 1$ is called the *duration* of W .

Throughout this section we denote by $sp(u, v, t)$ the duration of a shortest (i.e., having minimum arrival time) temporal walk in \mathcal{G} that starts at $u \in V(\mathcal{G})$ in timestep t and ends at $v \in V(\mathcal{G})$. (If $u = v$, $sp(u, v, t) = 0$.) We note that there is no guarantee that a walk between a pair of vertices u, v exists; in such cases we let $sp(u, v, t) = \infty$. The algorithms that we present in Sections 2.1 and 2.2 will repeatedly require us to compute such shortest walks for specific pairs of vertices $u, v \in V(\mathcal{G})$ and a timestep $t \in [L]$ – the following theorem allows us to do this:

► **Theorem 3** (Wu et al. [27]). *Let $\mathcal{G} = \langle G_1, \dots, G_L \rangle$ be an arbitrary temporal graph. Then, for any $u \in V(\mathcal{G})$ and $t \in [L]$, one can compute in $O(Ln^2)$ time for all $v \in V(\mathcal{G})$ a temporal walk that starts at u , ends at v and has duration $sp(u, v, t)$ (or determine that no such walk exists).*

The following two definitions will be used to describe the sets of candidate solutions for each of the problems that we consider in this section:

► **Definition 4** ((v, t, X) -tour). *A (v, t, X) -tour W in a given temporal graph \mathcal{G} is a strict temporal walk that starts at some vertex $v \in V(\mathcal{G})$ in timestep t and visits all vertices in $X \subseteq V(\mathcal{G})$. The arrival time $\alpha(W)$ of a (v, t, X) -tour W is the timestep after the timestep at the end of which W has for the first time visited all vertices in X .*

► **Definition 5** ((v, t, k) -tour). *A (v, t, k) -tour W in a given temporal graph \mathcal{G} is a (v, t, X) -tour for some subset $X \subseteq V(\mathcal{G})$ that satisfies $|X| = k$. The arrival time $\alpha(W)$ of a (v, t, k) -tour W is the timestep after the timestep at the end of which W has for the first time visited all vertices in X .*

A (v, t, X) -tour W ((v, t, k) -tour W^*) in a temporal graph \mathcal{G} is said to be *foremost* if $\alpha(W) \leq \alpha(W')$ ($\alpha(W^*) \leq \alpha(W^{*'})$) for any other (v, t, X) -tour W' (any other (v, t, k) -tour $W^{*'}$). We now formally define this section's two main problems of interest:

► **Definition 6** (K-FIXED TEXP). *An instance of the K-FIXED TEXP problem is given as a tuple (\mathcal{G}, s, X, k) where $\mathcal{G} = \langle G_1, \dots, G_L \rangle$ is an arbitrary temporal graph with underlying graph G and lifetime L ; s is a start vertex in $V(\mathcal{G})$; and $X \subseteq V(\mathcal{G})$ is a set of target vertices such that $|X| = k$. The problem then asks that we decide if there exists an $(s, 1, X)$ -tour W in \mathcal{G} .*

► **Definition 7** (K-ARBITRARY TEXP). *An instance of the K-ARBITRARY TEXP problem is given as a tuple (\mathcal{G}, s, k) where $\mathcal{G} = \langle G_1, \dots, G_L \rangle$ is an arbitrary temporal graph with underlying graph G and lifetime L ; s is a start vertex in $V(\mathcal{G})$; and $k \in \mathbb{N}$. The problem then asks that we decide whether there exists an $(s, 1, k)$ -tour W in \mathcal{G} .*

For yes-instances of K-FIXED TEXP or K-ARBITRARY TEXP, a tour with minimum arrival time (among all tours of the type sought) is called an *optimal solution*.

2.1 An FPT algorithm for k-fixed TEXP

In this section we provide a deterministic FPT-time algorithm for K-FIXED TEXP. Let (\mathcal{G}, s, X, k) be an instance of K-FIXED TEXP. Our algorithm looks for an earliest arrival time $(s, 1, X)$ -tour of \mathcal{G} via a dynamic programming (DP) approach. We note that the approach is essentially an adaptation of an algorithm proposed (independently by Bellman [5] and Held & Karp [18]) for the classic Travelling Salesperson Problem to the parameterized problem for temporal graphs.

► **Theorem 8.** *It is possible to decide any instance $I = (\mathcal{G}, s, X, k)$ of K-FIXED TEXP, and return an optimal solution if I is a yes-instance, in time $O(2^k k L n^2)$, where $n = |V(\mathcal{G})|$ and L is \mathcal{G} 's lifetime.*

Proof. First we describe our algorithm before proving its correctness and analysing its running time. We begin by specifying a dynamic programming formula for $F(S, v)$, by which we denote the minimum arrival time of any temporal walk in \mathcal{G} that starts at vertex $s \in V(\mathcal{G})$ in timestep 1, visits all vertices in $S \subseteq X$, and finishes at vertex $v \in S$. One can compute $F(S, v)$ via the following formula:

$$F(S, v) = \begin{cases} 1 + sp(s, v, 1) & (|S| = 1) \\ \min_{u \in S - \{v\}} [F(S - \{v\}, u) + sp(u, v, F(S - \{v\}, u))] & (|S| > 1) \end{cases} \quad (1)$$

Note that to compute $F(S, v)$ when $|S| > 1$, Equation (1) states that we need only consider values $F(S', u)$ with $u \in S'$ and $|S'| = |S| - 1$, and so we begin by computing all values $F(S', u)$ such that $S' \subseteq X$ satisfies $|S'| = 1$ and $u \in S'$, before computing all values such that $|S'| = 2$ and $u \in S'$ and so on, until we have computed all values $F(X, u)$ where $u \in X$ (i.e., values $F(S', u)$ with $|S'| = k = |X|$). Once all necessary values have been obtained, computing the following value gives the arrival time of an optimal $(s, 1, X)$ -tour:

$$F^* = \min_{v \in X} F(X, v). \quad (2)$$

If, whenever we compute a value $F(S, v)$ with $|S| > 1$, we also store alongside $F(S, v)$ a single pointer

$$p(S, v) = \arg \min_{u \in S - \{v\}} [F(S - \{v\}, u) + sp(u, v, F(S - \{v\}, u))],$$

then once we have computed F^* we can use a traceback procedure to reconstruct the walk with arrival time F^* . More specifically, let $u_1 = \arg \min_{u \in X} F(X, u)$ and $u_i = p(X - \{u_1, \dots, u_{i-2}\}, u_{i-1})$ for all $i \in [2, k]$. To complete the algorithm, we then check if F^* is finite: If so, then there must be a $(s, 1, X)$ -tour W in \mathcal{G} with $\alpha(W) = F^*$ that visits the vertices u_k, \dots, u_1 in that order. We can reconstruct W by concatenating the k shortest walks obtained by starting at s in timestep 1 and computing a shortest walk from s to u_k , then computing a shortest walk from u_k to u_{k-1} starting at the timestep at which u_k was reached, and so on, until u_1 is reached; once constructed, return W . If, on the other hand, $F^* = \infty$ (which is possible by the definition of $sp(u, v, t)$) then return no.

Correctness. The correctness of Equation (1) can be shown via induction on $|S|$: The base case (i.e., when $|S| = 1$) is correct since the arrival time of the foremost temporal walk that starts at s in timestep 1 and ends at a specific vertex $v \in X$ is clearly equal to one plus the duration of the foremost temporal walk between s and v starting at timestep 1.

15:6 Parameterized Temporal Exploration Problems

For the general case (when $|S| > 1$), assume first that the formula holds for any set S' such that $|S'| = l$ and any vertex $u \in S'$. To see that the formula holds for all sets S with $|S| = l + 1$ and vertices $v \in S$, consider any walk W that starts in timestep 1, visits all vertices in some set S with $|S| = l + 1$ and ends at v . Let x_1, \dots, x_{l+1} be the order in which the vertices $x_i \in S$ are reached by W for the first time; let $x = x_{l+1} = v$ and $x' = x_l$. Note that the subwalk W' of W that begins in timestep 1 and finishes at the end of the timestep in which W arrives at x' for the first time is surely an $(s, 1, S - \{v\})$ -tour, since W' visits every vertex in $S - \{x\} = S - \{v\}$. Then, by the induction hypothesis we have $\alpha(W') \geq F(S - \{v\}, x')$ because $|S - \{v\}| = l$, and since W ends at v we have

$$\begin{aligned} \alpha(W) &\geq \alpha(W') + sp(x', v, \alpha(W')) \\ &\geq F(S - \{v\}, x') + sp(x', v, F(S - \{v\}, x')). \end{aligned}$$

More generally, we can say that any $(s, 1, S)$ -tour W that starts at s in timestep 1, visits all vertices in S (where $|S| = l + 1$), and finishes at $v \in S$ satisfies the above inequality for some $x' \in S - \{v\}$. Note that for any $u \in S - \{v\}$, $F(S - \{v\}, u) + sp(u, v, F(S - \{v\}, u))$ corresponds to the arrival time of a valid $(s, 1, S)$ -tour, obtained by concatenating an earliest arrival time $(s, 1, S - \{v\})$ -tour that ends at u and a shortest walk between u and v starting at time $F(S - \{v\}, u)$. Therefore, to compute $F(S, v)$ it suffices to compute the minimum value of $F(S - \{v\}, u) + sp(u, v, F(S - \{v\}, u))$ over all $u \in S - \{v\}$; note that this is exactly Equation (1) in the case that $|S| > 1$.

To establish the correctness of Equation (2) recall that, by Definition 4, the arrival time of any $(s, 1, X)$ -tour in \mathcal{G} is equal to the timestep after the timestep in which it traverses a time edge to reach the final unvisited vertex of X for the first time. Assume that I is a yes-instance and let $x^* \in X$ be the k -th unique vertex in X that is visited by some foremost $(s, 1, X)$ -tour W ; then, by the analysis in the previous paragraph, we must have $\alpha(W) = F(X, x^*)$ since W is foremost, so $x^* = \arg \min_{v \in X} F(X, v)$ and thus $\alpha(W) = F(X, x^*) = \min_{v \in X} F(X, v) = F^*$, as required.

The fact that the answer returned by the algorithm is correct follows from the correctness of Equations (1) and (2) and the traceback procedure, together with the fact that I is a no-instance if and only if $F^* = \infty$. The details of this second claim are not difficult to see and are omitted, but we note that it is indeed possible that $F^* = \infty$ since F^* is the summation of a number of values $sp(u, v, t)$, some of which may satisfy $sp(u, v, t) = \infty$ by definition.

Runtime analysis. Since we only compute values of $F(S, v)$ such that $v \in S$ and $1 \leq |S| \leq k$, in total we compute $O(\sum_{i=1}^k \binom{k}{i} i) = O(2^k k)$ values. Note that, to compute any value $F(S, v)$ with $|S| = i > 1$, Equation (1) requires that we consider the values $F(S - \{v\}, u) + sp(u, v, F(S - \{v\}, u))$ with $u \in S - \{v\}$, of which there are exactly $i - 1$. We therefore use Theorem 3 to compute (and store temporarily), for each S' with $|S'| = i - 1$ and $x \in S'$, in $O(Ln^2)$ time the value of $sp(x, y, F(S', x))$ for all $y \in V(\mathcal{G})$ immediately after computing all $F(S', x)$, and use these precomputed shortest walk durations to compute $F(S, v)$ for any S with $|S| = i$ and $v \in S$ in time $O(i) = O(k)$. Thus, we spend $O(k) + O(Ln^2) = O(Ln^2)$ (since $k \leq n$) time for each of $O(2^k k)$ values $F(S, v)$. This yields an overall time of $O(2^k k Ln^2)$. Note that F^* can be computed using Equation (2) in $O(k)$ time since we take the minimum of $O(k)$ values; also note that a $(v, 1, X)$ -tour with arrival time F^* can be reconstructed in time $O(kLn^2)$ using the aforescribed traceback procedure, since we need to recompute $O(k)$ shortest walks, spending $O(Ln^2)$ time on each walk. Hence the overall running time of the algorithm is bounded by $O(2^k k Ln^2)$, as claimed. \blacktriangleleft

We remark that K -FIXED TEXP becomes TEXP if $X = V$, hence Theorem 8 also implies an FPT algorithm for TEXP parameterized by the number of vertices. Furthermore, we observe that TEXP is also FPT when parameterized by the lifetime L of the given temporal graph: If $L < n - 1$, the instance is clearly a no-instance, and if $L \geq n - 1$, the FPT algorithm for TEXP with parameter n is also FPT for parameter L .

2.2 FPT algorithms for k -arbitrary TEXP

The main result of this section is a randomized FPT-time algorithm for K -ARBITRARY TEXP that utilises the *colour-coding* technique originally presented by Alon, Yuster and Zwick [4]. There, they employed the technique primarily to detect the existence of a k -vertex simple path in a given undirected graph G . More generally, it has proven useful as a technique for finding fixed motifs (i.e., prespecified subgraphs) in static graphs/networks. We provide a high-level description of the technique and the way that we apply it at the beginning of Section 2.2.1. A standard derandomization technique (also originating from [4]) is then utilised within Section 2.2.2 to obtain a deterministic algorithm for K -ARBITRARY TEXP with a worse, but still FPT, running time.

2.2.1 A randomized algorithm

The algorithm of this section employs the colour-coding technique of Alon, Yuster and Zwick [4]. First, we informally sketch the structure of the algorithm behind Theorem 9: We colour the vertices of an input temporal graph uniformly at random, then by means of a DP subroutine we look for a temporal walk that begins at some start vertex s in timestep 1 and visits k vertices with distinct colours by the earliest time possible. Notice that if such a walk is found then it must be a (v, t, k) -tour, since the k vertices are distinctly coloured and therefore must be distinct. Then, the idea is to repeatedly: (1) randomly colour the input graph \mathcal{G} 's vertices; then (2) run the DP subroutine on each coloured version of \mathcal{G} . We repeat these steps enough times to ensure that, with high probability, the vertices of an optimal $(s, 1, k)$ -tour are coloured with distinct colours at least once over all colourings – if this happens then the DP subroutine will surely return an optimal $(s, 1, k)$ -tour or one with equal arrival time. With this high-level description in mind, we now present/analyse the algorithm:

► **Theorem 9.** *For every $\varepsilon > 0$, there exists a Monte Carlo algorithm that, with probability $1 - \varepsilon$, decides a given instance $I = (\mathcal{G}, s, k)$ of K -ARBITRARY TEXP , and returns an optimal solution if I is a yes-instance, in time $O((2e)^k L n^3 \log \frac{1}{\varepsilon})$, where $n = |V(\mathcal{G})|$ and L is \mathcal{G} 's lifetime.*

Proof. Let $V := V(\mathcal{G})$. We now describe our algorithm before proving it correct and analysing its running time. Let $c : V \rightarrow [k]$ be a colouring of the vertices $v \in V$. Let a walk W in \mathcal{G} that starts at s and visits a vertex coloured with each colour in $D \subseteq [k]$ be known as a D -colourful walk; let the timestep after the timestep at the end of which W has for the first time visited vertices with k distinct colours be known as the *arrival time* of W , denoted by $\alpha(W)$. The algorithm employs a subroutine that computes, should one exist, a $[k]$ -colourful walk W in \mathcal{G} with earliest arrival time. Note that a D -colourful walk ($D \subseteq [k]$) in \mathcal{G} is by definition an $(s, 1, |D|)$ -tour in \mathcal{G} .

15:8 Parameterized Temporal Exploration Problems

Define $H(D, v)$ to be the earliest arrival time of any D -colourful walk (where $D \subseteq [k]$) in \mathcal{G} that ends at a vertex v with $c(v) \in D$. The value of $H(D, v)$ for any $D \subseteq [k]$ and v with $c(v) \in D$ can be computed via the following dynamic programming formula (within the formula we denote by $D_{c(v)}^-$ the set $D - \{c(v)\}$):

$$H(D, v) = \begin{cases} 1 + sp(s, v, 1) & (|D| = 1) \\ \min_{u \in V: c(u) \in D_{c(v)}^-} [H(D_{c(v)}^-, u) + sp(u, v, H(D_{c(v)}^-, u))] & (|D| > 1) \end{cases} \quad (3)$$

In order to compute $H(D, v)$ for any $D \subseteq [k]$ and vertex v with $c(v) \in D$, Equation (3) requires that we consider values $H(D - \{c(v)\}, u)$ such that $c(u) \in D - \{c(v)\}$, and so we begin by computing $H(D', v)$ for all D' with $|D'| = 1$ and v with $c(v) \in D'$, then for all D' with $|D'| = 2$ and v with $c(v) \in D'$, and so on, until all values $H([k], v)$ have been obtained. The earliest arrival time of any $[k]$ -colourful walk in \mathcal{G} is then given by

$$H^* = \min_{u \in V(\mathcal{G})} H([k], u). \quad (4)$$

Once H^* has been computed, we check whether its value is finite or equal to ∞ . If H^* is finite then we can use a pointer system and traceback procedure (almost identical to those used in the proof of Theorem 8) to reconstruct an $(s, 1, k)$ -tour with arrival time H^* if one exists; otherwise we return no. This concludes the description of the dynamic programming subroutine.

Let $r = \lceil \frac{1}{\epsilon} \rceil$ and let W^* initially be the trivial walk that starts and finishes at vertex s in timestep 1. Perform the following two steps for $e^k \ln r$ iterations:

1. Assign colours in $[k]$ to the vertices of V uniformly at random and check if all k colours colour at least one vertex of G ; if not, start next iteration. If yes, proceed to step 2.
2. Run the DP subroutine in order to find an optimal $[k]$ -colourful walk W in \mathcal{G} if one exists. If such a W is found then check if $\alpha(W) < \alpha(W^*)$ or W^* starts and ends at s in timestep 1 (i.e., still has its initial value), and in either case set $W^* = W$; otherwise the DP subroutine returned no and we make no change to W^* .

Once all iterations of the above steps are over, check if W^* is still equal to the walk that starts and finishes at s in timestep 1; if not then return W^* , otherwise return no. This concludes the algorithm's description.

Correctness. We focus on proving the randomized aspect of the algorithm correct and omit correctness proofs for Equations (3) and (4) since the arguments are similar to those provided in Theorem 8's proof.

If I is a no-instance then in no iteration will the DP subroutine find an $(s, 1, k)$ -tour in \mathcal{G} . Hence in the final step the algorithm will find that W^* is equal to the walk that starts and ends at s in timestep 1 (by the correctness of Equations (3) and (4)) and return no, which is clearly correct. Assume then that I is yes-instance. Let W be an $(s, 1, k)$ -tour in \mathcal{G} with earliest arrival time, and let $X \subseteq V$ be the set of k vertices visited by W . Then, if during one of the $e^k \ln r$ iterations of steps 1 and 2 we colour the vertices of V in such a way that X is well-coloured (we say that a set of vertices $U \subseteq V$ is *well-coloured* by colouring c if $c(u) \neq c(v)$ for every pair of vertices $u, v \in U$), W will induce an optimal $[k]$ -colourful walk in \mathcal{G} . The DP subroutine will then return W or some other optimal $[k]$ -colourful walk W' with $\alpha(W) = \alpha(W')$ that visits a well-coloured subset of vertices X' ; note that the arrival time of the best tour found in any iteration so far will then surely be $\alpha(W)$, since W has earliest arrival time.

Observe that if we colour the vertices of V with k colours uniformly at random, then, since $|X| = k$, there are k^k ways to colour the vertices in $X \subseteq V$, of which $k!$ constitute well-colourings of X . Hence after a single colouring of V we have

$$\Pr[X \text{ is well-coloured}] = \frac{k!}{k^k} > \frac{1}{e^k},$$

where the inequality follows from the fact that $k!/k^k > \sqrt{2\pi k}^{\frac{1}{2}} e^{\frac{1}{12k+1}} / e^k$ (this inequality is due to Robbins [25] and is related to Stirling's formula). Hence, after $e^k \ln r$ colourings, we have (using the standard inequality $(1 - \frac{1}{x})^x \leq \frac{1}{e}$ for all $x \geq 1$):

$$\Pr[X \text{ is not well-coloured in any colouring}] \leq \left(1 - \frac{1}{e^k}\right)^{e^k \ln r} \leq 1/r \leq \varepsilon.$$

Thus, the probability that X is well-coloured at least once after $e^k \ln r$ colourings is at least $1 - \varepsilon$. It follows that, with probability $\geq 1 - \varepsilon$, the earliest arrival $[k]$ -colourful walk returned by the algorithm after all iterations is in fact an optimal $(s, 1, k)$ -tour in \mathcal{G} , since either W or some other $(s, 1, k)$ -tour with equal arrival time will eventually be returned.

Runtime analysis. Note that the DP subroutine computes exactly the values $H(D, v)$ such that $D \subseteq [k]$ and v satisfies $c(v) \in D$. Hence there are at most $\binom{k}{i}n$ values $H(D, v)$ such that $|D| = i$, for all $i \in [k]$; this gives a total of $\sum_{i \in [k]} \binom{k}{i}n = O(2^k n)$ values. In order to compute $H(D, v)$ for any D with $|D| = i > 1$, Equation (3) requires us to consider the value of $H(D - \{c(v)\}, u) + sp(u, v, H(D - \{c(v)\}, u))$ for all u such that $c(u) \in D - \{c(v)\}$. Therefore, similar to the algorithm in the proof of Theorem 8, we compute and store, immediately after computing each value $H(D', x)$ with $|D'| = i - 1$ and $c(x) \in D'$, the value of $sp(x, y, H(D', x))$ for all $y \in V(\mathcal{G})$ in $O(Ln^2)$ time (Theorem 3). Note that there can be at most n vertices u such that $c(u) \in D - \{c(v)\}$, and so in total we spend $O(n) + O(Ln^2) = O(Ln^2)$ time on each of $O(2^k n)$ values of $H(D, v)$, giving an overall time of $O(2^k Ln^3)$. We can compute H^* in $O(n)$ time since we take the minimum of $O(n)$ values, and the traceback procedure can be performed in $O(kLn^2) = O(Ln^3)$ time since we concatenate k walks obtained using Theorem 3. Thus the overall time spent carrying out one execution of the DP subroutine is $O(2^k Ln^3)$.

Since the running time of each iteration of the main algorithm is dominated by the running time of the DP subroutine and there are $e^k \ln r = O(e^k \log \frac{1}{\varepsilon})$ iterations in total, we conclude that the overall running time of the algorithm is $O((2e)^k Ln^3 \log \frac{1}{\varepsilon})$, as claimed. This completes the proof. \blacktriangleleft

2.2.2 Derandomizing the algorithm of Theorem 9

The randomized colour-coding algorithm of Theorem 9 can be derandomized at the expense of incurring a $k^{O(\log k)} \log n$ factor in the running time. We employ a standard derandomization technique, presented initially in [4], which involves the enumeration of a k -perfect family of hash functions from $[n]$ to $[k]$. The functions in such a family will be viewed as colourings of the vertex set of the temporal graph given as input to the k -ARBITRARY TEXP problem.

Formally, a family \mathcal{H} of hash functions from $[n]$ to $[k]$ is k -perfect if, for every subset $S \subseteq [n]$ with $|S| = k$, there exists a function $f \in \mathcal{H}$ such that f restricted to S is bijective (i.e., one-to-one). The following theorem of Naor et al. enables one to construct such a family \mathcal{H} in time linear in the size of \mathcal{H} :

15:10 Parameterized Temporal Exploration Problems

► **Theorem 10** (Naor, Schulman and Srinivasan [24]). *A k -perfect family \mathcal{H} of hash functions f_i from $[n]$ to $[k]$, with size $e^k k^{O(\log k)} \log n$, can be computed in $e^k k^{O(\log k)} \log n$ -time.*

We note that the value of $f_i(x)$ for any $f_i \in \mathcal{H}$ and $x \in [n]$ can be evaluated in $O(1)$ time.

To solve an instance of K -ARBITRARY TEXP, we can now use the algorithm from the proof of Theorem 9, but instead of iterating over $e^k \ln r$ random colourings, we iterate over the $e^k k^{O(\log k)} \log n$ hash functions in the k -perfect family of hash functions constructed using Theorem 10. This ensures that the set X of k vertices visited by an optimal $(s, 1, k)$ -tour is well-coloured in at least one iteration, and we obtain the following theorem.

► **Theorem 11.** *There is a deterministic algorithm that can solve a given instance (\mathcal{G}, s, k) of K -ARBITRARY TEXP in $(2e)^k k^{O(\log k)} Ln^3 \log n$ time, where $n = |V(\mathcal{G})|$. If the instance is a yes-instance, the algorithm also returns an optimal solution.*

We remark that, since a temporal walk can visit at most $L + 1$ vertices in a temporal graph with lifetime L , Theorem 11 also implies an FPT algorithm for the following problem, parameterized by the lifetime L of the given temporal graph: Find a temporal walk that visits as many distinct vertices as possible.

3 Non-Strict TEXP parameterizations

In this section we consider the non-strict version of TEXP, in which a walk is allowed to traverse an unlimited number of edges in every timestep. As mentioned in the introduction, this changes the nature of the problem significantly. In particular, it means that a temporal walk positioned at a vertex v in timestep t is able to visit, during timestep t , any other vertex contained in the same connected component C as v and move to an arbitrary vertex $u \in C$, beginning timestep $t + 1$ positioned at vertex u . As such, it is no longer necessary to know the edge structure of the input temporal graph during each timestep, and we can focus only on the connected components of each layer. This leads to the following definition:

► **Definition 12** (Non-strict temporal graph, \mathcal{G}). *A non-strict temporal graph $\mathcal{G} = \langle G_1, \dots, G_L \rangle$ with vertex set $V := V(\mathcal{G})$ and lifetime L is an indexed sequence of partitions (layers) $G_t = \{C_{t,1}, \dots, C_{t,s_t}\}$ of V for $t \in [L]$. For all $t \in [L]$, each $v \in V$ satisfies $v \in C_{t,j}$ for a unique $j \in [s_t]$. The integer s_t denotes the number of components in layer G_t ; clearly we have $s_t \in [n]$.*

A non-strict temporal walk is then defined as follows:

► **Definition 13** (Non-strict temporal walk, W). *A non-strict temporal walk W starting at vertex v at time t_1 in a non-strict temporal graph $\mathcal{G} = \langle G_1, \dots, G_L \rangle$ is a sequence $W = C_{t_1, j_1}, C_{t_2, j_2}, \dots, C_{t_l, j_l}$ of components C_{t_i, j_i} ($i \in [l]$) with $1 \leq t_1 \leq t_l \leq L$ such that: $t_i + 1 = t_{i+1}$ for all $i \in [1, l - 1]$; $C_{t_i, j_i} \in G_{t_i}$ and $j_i \in [s_{t_i}]$ for all $i \in [l]$; $C_{t_i, j_i} \cap C_{t_{i+1}, j_{i+1}} \neq \emptyset$ for all $i \in [l - 1]$; and $v \in C_{t_1, j_1}$.*

Let $W = C_{t_1, j_1}, C_{t_2, j_2}, \dots, C_{t_l, j_l}$ be a non-strict temporal walk in some non-strict temporal graph \mathcal{G} starting at some vertex $s \in C_{t_1, j_1}$. We call $l \in [L]$ the *duration* of W . The walk W is said to start at vertex $s \in C_{t_1, j_1}$ in timestep t_1 and finish at component C_{t_l, j_l} (or sometimes at some $v \in C_{t_l, j_l}$) in timestep t_l . Furthermore, W *visits* the set of vertices $\bigcup_{i \in [l]} C_{t_i, j_i}$. Note that W visits exactly one component in each of the l timesteps that make up its duration. We call W *non-strict exploration schedule starting at s with arrival time l* if $t_1 = 1$ and $\bigcup_{i \in [l]} C_{t_i, j_i} = V(\mathcal{G})$. As FPT algorithms for K -FIXED TEXP and K -ARBITRARY TEXP for non-strict temporal graphs can be derived using similar techniques as in Section 2, we instead consider the following two non-strict exploration problems in this section:

► **Definition 14** (NON-STRICT TEMPORAL EXPLORATION (NS-TEXP)). *An instance of NS-TEXP is given as a tuple (\mathcal{G}, s) , where \mathcal{G} is a non-strict temporal graph with lifetime L and $s \in V(\mathcal{G})$ is a start vertex. The problem then asks whether or not \mathcal{G} admits an exploration schedule that starts at s .*

► **Definition 15** (SET NS-TEXP). *An instance of SET NS-TEXP is given as a tuple $(\mathcal{G}, s, \mathcal{X})$, where \mathcal{G} is a non-strict temporal graph with lifetime L , $s \in V(\mathcal{G})$ is a start vertex, and $\mathcal{X} = \{X_1, \dots, X_m\}$ is a set of subsets $X_i \subseteq V(\mathcal{G})$. The problem then asks whether or not there exists a non-strict temporal walk in \mathcal{G} that starts at s in timestep 1 and visits at least one vertex contained in X_i for all $i \in [m]$.*

In the following two subsections we establish FPT-membership for NS-TEXP when parameterized by the lifetime L , then prove W[2]-hardness for the SET NS-TEXP problem when the same parameter is considered.

3.1 An FPT algorithm for NS-TEXP with parameter L

Let NS-TEXP-L be the variant of NS-TEXP parameterized by the lifetime L of the input temporal graph \mathcal{G} ; let an instance of NS-TEXP-L be given as a tuple (\mathcal{G}, s, L) . We prove that NS-TEXP-L \in FPT by specifying a bounded search tree-based algorithm.

Let $\mathcal{G} = \langle G_1, \dots, G_L \rangle$ be some non-strict temporal graph. Throughout this section we let $\mathcal{C}(\mathcal{G}) := \bigcup_{t \in [L]} G_t$, i.e., $\mathcal{C}(\mathcal{G})$ is the set of all components belonging to some layer of \mathcal{G} . We implicitly assume that each component $C \in \mathcal{C}(\mathcal{G})$ is *associated* with a unique layer G_t of \mathcal{G} in which it is contained. If a component (seen as just a set of vertices) occurs in several layers, we thus treat these occurrences as different elements of $\mathcal{C}(\mathcal{G})$ (or of any subset thereof) because they are associated with different layers. If X is a set of components in $\mathcal{C}(\mathcal{G})$ that are associated with distinct layers (i.e., no two components in X are associated with the same layer G_t of \mathcal{G}), then we say that the components in X *originate from unique layers of \mathcal{G}* . For a set X of components that originate from unique layers of \mathcal{G} , we let $D(X) := \bigcup_{C \in X} C$ be the union of the vertex sets of the components in X . For any such set X , we also let $T(X) = \{t \in [L] : \text{there is a } C \in X \text{ associated with layer } G_t\}$.

Within the following, we assume that \mathcal{G} admits a non-strict exploration schedule W :

► **Observation 16.** *Let X ($|X| \in [0, L - 1]$) be a subset of the components visited by the exploration schedule W . Then there exists $C \in \mathcal{C}(\mathcal{G}) - X$ with $C \in G_t$ ($t \in [L] - T(X)$) such that $|C - D(X)| \geq (n - |D(X)|)/(L - |T(X)|)$.*

Observation 16 follows since, otherwise, W visits at most $L - |T(X)|$ components $C \in \mathcal{C}(\mathcal{G}) - X$ that each contain $|C - D(X)| < (n - |D(X)|)/(L - |T(X)|)$ of the vertices $v \notin D(X)$, and so the total number of vertices visited by W is strictly less than $|D(X)| + (L - |T(X)|) \cdot (n - |D(X)|)/(L - |T(X)|) = n$, a contradiction.

We briefly outline the main idea of our FPT result: We use a search tree algorithm that maintains a set X of components that a potential exploration schedule could visit, starting with the empty set. Then the algorithm repeatedly tries all possibilities for adding a component (from some so far untouched layer) that contains at least $(n - |D(X)|)/(L - |T(X)|)$ unvisited vertices (whose existence is guaranteed by Observation 16 if there exists an exploration schedule). It is clear that the search tree has depth L , and the main further ingredient is an argument showing that the number of candidates for the component to be added is bounded by a function of L , namely, by $(L - |T(X)|)^2$: This is because each of the $L - |T(X)|$ untouched layers can contain at most $L - |T(X)|$ components that each contain at least $(n - |D(X)|)/(L - |T(X)|)$ unvisited vertices. We now proceed to describe the details of the algorithm and its analysis.

15:12 Parameterized Temporal Exploration Problems

► **Lemma 17.** *Let $\mathcal{G} = \langle G_1, \dots, G_L \rangle$ be an arbitrary order- n non-strict temporal graph. Then, for components $C_{t_1, j_1} \in G_{t_1}$ and $C_{t_2, j_2} \in G_{t_2}$ (with $1 \leq t_1 \leq t_2 \leq L$) one can decide, in $O((t_2 - t_1 + 1)n)$ time, whether there exists a non-strict temporal walk beginning at any vertex contained in C_{t_1, j_1} in timestep t_1 and finishing at C_{t_2, j_2} in timestep t_2 .*

Proof. For any $v \in V(\mathcal{G})$ and $t \in [t_1, t_2]$, let $c(v, t)$ denote the component $C_{t, j}$ such that $v \in C_{t, j}$ during timestep t . First, precompute the values $c(v, t)$ by, for every $t \in [t_1, t_2]$, scanning each component $C \in G_t$ and setting $c(v, t) = C$ if and only if $v \in C$. Next, let $X_{t_1} = \{C_{t_1, j_1}\}$ and then consider the timesteps $t \in [t_1 + 1, t_2]$ in increasing order, constructing at each timestep t the set $X_t = X_{t-1} \cup \{c(v, t) : v \in \bigcup_{C \in X_{t-1}} C\}$. Finally, check whether $C_{t_2, j_2} \in X_{t_2}$, returning yes if so and no otherwise.

The correctness of the algorithm is not hard to see. To see that the claimed running time of $O((t_2 - t_1 + 1)n)$ holds, note first that precomputing the values $c(v, t)$ for any $v \in V(\mathcal{G})$ and any $t \in [t_1, t_2]$ requires $O((t_2 - t_1 + 1)n)$ time since, in each timestep $t \in [t_1, t_2]$, we simply iterate over the vertices (of which there are always n in total) contained in each component $C \in G_t$. Then, to compute X_t for each $t \in [t_1 + 1, t_2]$, we add $c(v, t)$ (which can be evaluated in $O(1)$ time due to our preprocessing step) to X_t for each vertex $v \in \bigcup_{C \in X_{t-1}} C$, of which there can be at most n . This second step of the algorithm also clearly requires $O((t_2 - t_1 + 1)n)$ time, and the lemma follows. ◀

Let X be a set of components originating from unique layers of \mathcal{G} , and let $W_{\mathcal{G}}^?(s, X) = \text{yes}$ if and only if there exists a non-strict temporal walk in \mathcal{G} that starts at $s \in V(\mathcal{G})$ in timestep 1 and visits at least the components contained in X , and no otherwise.

► **Lemma 18.** *For any order- n non-strict temporal graph $\mathcal{G} = \langle G_1, \dots, G_L \rangle$, any $s \in V(\mathcal{G})$, and any set X of components originating from unique layers of \mathcal{G} , $W_{\mathcal{G}}^?(s, X)$ can be computed in $O(Ln)$ time.*

Proof. Let $C_{s_1}, C_{s_2}, \dots, C_{s_{|X|}}$ be an index-ordered sequence of the components in X , with the indices $s_i \in [L]$ satisfying $C_{s_i} \in G_{s_i}$ (for all $i \in [|X|]$) and $s_i < s_{i+1}$ (for all $i \in [|X| - 1]$). Let $C_s \in G_1$ be the unique component in layer 1 such that $s \in C_s$ (note that we may have $C_{s_1} = C_s$). Now, apply the algorithm of Lemma 17 with $C_{t_1, j_1} = C_s$ and $C_{t_2, j_2} = C_{s_1}$, and then with $C_{t_1, j_1} = C_{s_i}$ and $C_{t_2, j_2} = C_{s_{i+1}}$ for all $i \in [|X| - 1]$. If the return value of any application of the algorithm of Lemma 17 is no, then we return $W_{\mathcal{G}}^?(s, X) = \text{no}$; otherwise we return $W_{\mathcal{G}}^?(s, X) = \text{yes}$. This concludes the algorithm's description.

Since each component C_{s_i} can only be visited in timestep s_i it is clear that any walk that visits all components of X must visit them in the specified order. The algorithm sets $W_{\mathcal{G}}^?(s, X) = \text{yes}$ if the components of X can be visited in the specified order. On the other hand, if Lemma 17 returns no for at least one pair of input components $C_{s_i}, C_{s_{i+1}}$ (or C_s, C_{s_1}), then it must be that the components cannot be visited in this order, and thus the algorithm sets $W_{\mathcal{G}}^?(s, X) = \text{no}$. Thus, the algorithm's correctness follows from the correctness of Lemma 17's algorithm. To see that the runtime of the algorithm is bounded by $O(Ln)$, recall that each application of Lemma 17's algorithm to start/finish components C_{s_i} and $C_{s_{i+1}}$ takes $c(s_{i+1} - s_i + 1)n$ time (for a constant c hidden in the bound of Lemma 17). Thus the total amount of time spent over all applications is $c(s_1 - 1 + 1)n + \sum_{i \in [|X| - 1]} c(s_{i+1} - s_i + 1)n = cn(s_{|X|} + |X| - 1) \leq cn(2L - 1) = O(Ln)$, where the last inequality holds since $|X|, s_{|X|} \leq L$. ◀

Now, let \mathcal{G} be some input graph, and let X be some set of components originating from unique layers of \mathcal{G} . For any $s \in V(\mathcal{G})$, the recursive function $g(\mathcal{G}, s, X)$ (Algorithm 1) returns yes if and only if there exists a non-strict exploration schedule of \mathcal{G} that starts at s and visits (at least) the components contained in X , and returns no otherwise. We prove the correctness of Algorithm 1 in Lemma 19.

■ **Algorithm 1** Recursive function $g(\mathcal{G}, s, X)$.

```

1 if  $|X| = L$  or  $|D(X)| = n$  then
2   | if  $|D(X)| = n$  then return  $W_{\mathcal{G}}^?(s, X)$ 
3   | else return no
4 else
5   |  $C' \leftarrow \{C \in \mathcal{C}(\mathcal{G}) - X : |C - D(X)| \geq (n - |D(X)|)/(L - |T(X)|)\}$ 
6   |  $C^* \leftarrow C' - \{C \in C' : C \in G_t, t \in T(X)\}$ .
7   | if  $|C^*| = 0$  then return no
8   | for  $C \in C^*$  do
9   |   | if  $g(\mathcal{G}, s, X \cup \{C\}) = \text{yes}$  then return yes
10  | end
11  | return no
12 end
```

► **Lemma 19.** *For any non-strict temporal graph \mathcal{G} , any $s \in V(\mathcal{G})$, and any set X (with $|X| \in [0, L]$) containing components originating from unique layers of \mathcal{G} , Algorithm 1 correctly computes $g(\mathcal{G}, s, X)$.*

Proof. We first show that $g(\mathcal{G}, s, X)$ is correct in the base case, i.e., when $|X| = L$ or $|D(X)| = n$. If we have $|D(X)| = n$, then any non-strict temporal walk that starts at s in timestep 1 and visits all components in X is an exploration schedule. Thus, the correctness of line 2 follows from the definition of the return value $W_{\mathcal{G}}^?(s, X)$ (which can be computed using Lemma 18). If $|X| = L$ and $|D(X)| < n$, i.e., we have reached line 3, then there must exist no exploration schedule that visits each of the components in X , since any non-strict temporal walk of duration at most L can visit at most L components, but at least one additional component $C \notin X$ needs to be visited to cover at least one vertex $v \notin D(X)$ – thus it is correct to return *no* in this case.

Otherwise, we have $|X| < L$ and $|D(X)| < n$, and are in the recursive case. Then, by Observation 16, any non-strict exploration schedule that visits all components in X must visit at least one other component $C \in \mathcal{C}(\mathcal{G}) - X$ such that $|C - D(X)| \geq (n - |D(X)|)/(L - |T(X)|)$. Line 5 computes the set C' consisting of all such components, line 6 forms from C' the set C^* by removing from C' any components that originate from layers t such that $C \in G_t$ for some $C \in X$ (since only one component can be visited in each timestep, and thus we want X to be a set of components originating from unique layers of \mathcal{G}). We remark that a more efficient implementation could skip layers G_t with $t \in T(X)$ already when constructing C' in line 5, but the asymptotic running-time of the overall algorithm would not be affected by this change. The correctness of line 7 follows from Observation 16. To complete the proof, we claim that the value *yes* is returned by line 9 if and only if there exists a non-strict temporal exploration schedule starting at s that visits all the components contained in X ; we proceed by reverse induction on $|X|$. Assume first that the return value of $g(\mathcal{G}, s, X')$ is correct for any X' with $|X'| = k$ ($k \in [L]$) and let $|X| = k - 1$. Now assume that, during the execution of $g(\mathcal{G}, s, X)$, line 9 returns *yes*; it follows that $g(\mathcal{G}, s, X') = \text{yes}$ for some $X' = X \cup C$ with $C \in C^*$ and thus it follows from the induction hypothesis that there exists a non-strict temporal exploration schedule that starts at s and visits all the components contained in X , as required. In the other direction, assume that there exists some non-strict exploration schedule W that starts at s in timestep 1 and visits all the components in X . Note that, since the execution has reached line 9, we surely have $|C^*| > 0$; since we also have $|X| < L$ and

$|D(X)| < n$ it follows from Observation 16 that W visits at least one additional component $C \in C^*$. Then, by the induction hypothesis, we must have $g(\mathcal{G}, s, X \cup \{C\}) = \text{yes}$; thus when the loop of lines 8–10 processes $C \in C^*$ the algorithm will return yes as required. ◀

► **Theorem 20.** *It is possible to decide any instance $I = (\mathcal{G}, s, L)$ of NS-TEXP- L in $O(L(L!)^2n)$ time.*

Proof. The algorithm simply returns the value of function call $g(\mathcal{G}, s, \emptyset)$ (Algorithm 1).

By Lemma 19, $g(\mathcal{G}, v, X)$ returns yes if and only if \mathcal{G} admits a non-strict exploration schedule that starts at v and visits at least the components contained in the set X (which contains $|X| \in [0, L]$ components originating from unique layers of \mathcal{G}), and returns no otherwise. Thus the correctness of the above follows immediately.

In order to bound the running time of the above algorithm, it suffices to bound the running time of Algorithm 1, i.e., the recursive function g . The initial call is $g(\mathcal{G}, s, \emptyset)$, and each recursive call is of the form $g(\mathcal{G}, s, X)$ where X is a set of components with size one more than the input set of the parent call. Hence, line 1 ensures that there are at most L levels of recursion in total (not including the level containing the initial call). For a call at level $i \geq 0$, the set C^* constructed in line 5 has size at most $(L - i)^2$, since at most $L - i$ components can cover at least $(n - |D(X)|)/(L - i)$ of the vertices in $V(\mathcal{G}) - D(X)$ during each of the $L - i$ steps $t \in [L] - T(X)$. Thus each call at level $i \geq 0$ makes at most $(L - i)^2$ recursive calls. The tree of recursive calls thus has at most $(L!)^2$ nodes at depth L , and hence $O((L!)^2)$ nodes in total. It follows that the overall number of calls is bounded by $O((L!)^2)$.

Next, note that if some level- i call $g(\mathcal{G}, s, X)$ is such that $|X| < L$ and $|D(X)| < n$, then line 5 computes the set C' , which can be achieved in $O(Ln)$ time by, for each $t \in [L]$, scanning over the components $C \in G_t$ (which collectively contain n vertices) and adding a component $C \in G_t$ to C' if and only if $|C - D(X)| \geq (n - |D(X)|)/(L - i)$. (Note that we can maintain a map from V to $\{0, 1\}$ that records for each vertex v whether $v \in D(X)$, and hence the value $|C - D(X)|$ can be computed in $O(|C|)$ time.) To compute the set C^* in line 6 we can follow a similar approach: for each $t \in [L] - T(X)$ ($|[L] - T(X)| = L - i$), add a component $C \in G_t$ to C^* if and only if it satisfies $C \in C'$. This requires $O((L - i)n) = O(Ln)$ time, and thus lines 5–6 take $O(Ln)$ time in total. Additionally, the return value of each recursive call is checked by the foreach loop (line 9) of its parent call in $O(1)$ time – this contributes an extra $O((L!)^2)$ time over all recursive calls. On the other hand, if a call $g(\mathcal{G}, s, X)$ is such that $|X| = L$ or $|D(X)| = n$, then line 2 computes $W_{\mathcal{G}}^2(s, X)$ in $O(Ln)$ time using Lemma 18. Thus in all cases the overall work per recursive call is $O(Ln)$, and the total amount of time spent before $g(\mathcal{G}, s, \emptyset)$ is returned is $O((L!)^2) \cdot O(Ln) = O(L(L!)^2n)$, as claimed. ◀

3.2 W[2]-hardness of Set NS-TEXP for parameter L

Our aim in this section is to show that the SET NS-TEXP problem is W[2]-hard when parameterized by the lifetime L of the input graph. The reduction is from the well-known SET COVER problem with parameter k – the maximum number of sets allowed in a candidate solution. SET COVER is known to be W[2]-hard for this parameterization [7].

► **Definition 21** (SET COVER). *An instance of SET COVER is given as a tuple (U, \mathcal{S}, k) , where $U = \{a_1, \dots, a_n\}$ is the ground set and $\mathcal{S} = \{S_1, \dots, S_m\}$ is a set of subsets $S_i \subseteq U$. The problem then asks whether or not there exists a subset $\mathcal{S}' \subseteq \mathcal{S}$ of size at most k such that, for all $i \in [n]$, there exists an $S \in \mathcal{S}'$ such that $a_i \in S$.*

For any instance I of SET COVER that we consider, we will w.l.o.g. assume that for each $i \in [n]$ we have $a_i \in S_j$ for some $j \in [m]$.

► **Theorem 22.** *SET NS-TEXP parameterized by L (the lifetime of the input non-strict temporal graph) is $W[2]$ -hard.*

Proof. Let $I = (U = \{a_1, \dots, a_n\}, \mathcal{S} = \{S_1, \dots, S_m\}, k)$ be an arbitrary instance of SET COVER parameterized by k . We construct a corresponding instance $I' = (\mathcal{G}, s, \mathcal{X})$ of SET NS-TEXP as follows: Let $V(\mathcal{G}) = \{s\} \cup \{x_j : j \in [m]\} \cup \{y_{i,j} : j \in [m], a_i \in S_j\}$, and define $X_i = \{y_{i,j} \in V(\mathcal{G}) : j \in [m]\}$ ($i \in [n]$) and $\mathcal{X} = \bigcup_{i \in [n]} \{X_i\}$. We set the lifetime L of \mathcal{G} to $L = 2k$ and specify the components for each timestep $t \in [2k]$ as follows: In all odd steps let one component be $\{s\} \cup \{x_j : j \in [m]\}$ and let all other vertices belong to components of size 1. In even steps, for each $j \in [m]$ let there be a component $\{y_{i,j} \in V(\mathcal{G}) : i \in [n]\} \cup \{x_j\}$ and let s form a component of size 1. Since $|V(\mathcal{G})| \leq 1 + m + mn = O(mn)$, $|\bigcup_{i \in [n]} X_i| = O(mn)$ and $L = 2k$ we have that the size of instance I' is $|I'| = O(kmn)$ and the parameter L is bounded solely by a function of instance I 's parameter k , as required. To complete the proof, we argue that I is a yes-instance if and only if I' is a yes-instance:

(\implies) Assume that I is a yes-instance; then there exists a collection of sets $\mathcal{S}' \subseteq \mathcal{S}$ of size $|\mathcal{S}'| = k' \leq k$ and, for all $i \in [n]$, there exists $S \in \mathcal{S}'$ with $a_i \in S$. Let $S_{j_1}, S_{j_2}, \dots, S_{j_{k'}}$ be an arbitrary ordering of the sets in \mathcal{S}' ; note that $j_i \leq m$ for all $i \in [k']$. We construct a non-strict temporal walk W in \mathcal{G} as follows: Starting at vertex s , for every $l \in [1, k']$, during timestep $t = 2l - 1$ visit all vertices in the current component then finish timestep $2l - 1$ positioned at x_{j_l} . The component occupied during step $2l$ will be the one containing x_{j_l} – explore all vertices contained in that component and finish step $2l$ positioned at x_{j_l} . If $k' < k$, then spend the steps of the interval $[2k' + 1, 2k]$ positioned in an arbitrary component. We claim that W visits at least one vertex in X_i for all $i \in [n]$. To see this, first note that for every $i \in [n]$ there exists an $S_j \in \mathcal{S}'$ such that $a_i \in S_j$. Hence, by our reduction, it follows that a vertex $y_{i,j}$ is contained in the component containing x_j during timestep $2l$ for every $l \in [k]$ and, by its construction, W visits the component containing x_j (and thus visits $y_{i,j} \in X_i$) during timestep $2l^*$ for some l^* such that $j_{l^*} = j$. Since this holds for all $i \in [n]$ it follows that W is a feasible solution and I' is a yes-instance.

(\impliedby) Assume that I' is a yes-instance and that we have some non-strict temporal walk W that visits at least one vertex in X_i for all $i \in [n]$. We first claim that W visits any vertex of the form $y_{i,j}$ for the first time during an even step. To see this, observe that every $y_{i,j}$ lies disconnected in its own component in every odd step t , and so to visit any $y_{i,j}$ in an odd step W would need to occupy the component containing $y_{i,j}$ during step $t - 1$ and finish step $t - 1$ positioned at $y_{i,j}$; hence $y_{i,j}$ was already visited in step $t - 1$, which is even. Therefore, in order for W to visit any $y_{i,j}$ it must be positioned, during at least one even step, at the component containing x_j . Now, to construct a collection of subsets $\mathcal{S}' \subseteq \mathcal{S}$ with size $x \leq k$, let $\mathcal{S}' = \{S_j : W \text{ visits the component containing } x_j \text{ during some even timestep}\}$. To see that \mathcal{S}' is a cover of U with size $x \leq k$, observe that W visits at least one vertex $y_{i,j}$ for every $i \in [n]$; thus, by the reduction, for every $i \in [n]$ the element a_i is contained in set S_j for some $S_j \in \mathcal{S}'$. It follows that the union of \mathcal{S}' 's elements covers U , and so I is a yes-instance. ◀

4 Conclusion

In this paper we have initiated the study of temporal exploration problems from the viewpoint of parameterized complexity. For both strict and non-strict temporal walks, we have shown several variants of the exploration problem to be in FPT. For the variant where we are given a family of vertex subsets and need to visit only one vertex from each subset, we have shown $W[2]$ -hardness for the non-strict model for parameter L . (In the strict model, one can show that $W[2]$ -hardness holds for this problem even in the case where each layer of the temporal graph is a complete graph.) An interesting question for future work is whether NS-TEXP is in FPT if the parameter is the maximum number of components in any layer.

References

- 1 Eleni C. Akrida, Jurek Czyzowicz, Leszek Gąsieniec, Łukasz Kuszner, and Paul G. Spirakis. Temporal flows in temporal networks. *Journal of Computer and System Sciences*, 103:46–60, 2019. doi:10.1016/j.jcss.2019.02.003.
- 2 Eleni C. Akrida, George B. Mertzios, and Paul G. Spirakis. The temporal explorer who returns to the base. In Pinar Heggernes, editor, *11th International Conference on Algorithms and Complexity (CIAC 2019)*, volume 11485 of *Lecture Notes in Computer Science*, pages 13–24. Springer, 2019. doi:10.1007/978-3-030-17402-6_2.
- 3 Eleni C. Akrida, George B. Mertzios, Paul G. Spirakis, and Viktor Zamaraev. Temporal vertex cover with a sliding time window. *Journal of Computer and System Sciences*, 107:108–123, 2020. doi:10.1016/j.jcss.2019.08.002.
- 4 Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *Journal of the ACM*, 42(4):844–856, July 1995. doi:10.1145/210332.210337.
- 5 Richard Bellman. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM*, 9(1):61–63, January 1962. doi:10.1145/321105.321111.
- 6 Hans L. Bodlaender and Tom C. van der Zanden. On exploring always-connected temporal graphs of small pathwidth. *Information Processing Letters*, 142:68–71, 2019. doi:10.1016/j.ipl.2018.10.016.
- 7 Bonnet, Édouard, Paschos, Vangelis Th., and Sikora, Florian. Parameterized exact and approximation algorithms for maximum k-set cover and related satisfiability problems. *RAIRO-Theoretical Informatics and Applications*, 50(3):227–240, 2016. doi:10.1051/ita/2016022.
- 8 Björn Brodén, Mikael Hammar, and Bengt J. Nilsson. Online and offline algorithms for the time-dependent TSP with time zones. *Algorithmica*, 39(4):299–319, 2004. doi:10.1007/s00453-004-1088-z.
- 9 Binh-Minh Bui-Xuan, Afonso Ferreira, and Aubin Jarry. Computing shortest, fastest, and foremost journeys in dynamic networks. *International Journal of Foundations of Computer Science*, 14(2):267–285, 2003. doi:10.1142/S0129054103001728.
- 10 Benjamin Merlin Bumpus and Kitty Meeks. Edge exploration of temporal graphs. In Paola Flocchini and Lucia Moura, editors, *32nd International Workshop on Combinatorial Algorithms (IWCOA 2021)*, volume 12757 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2021. doi:10.1007/978-3-030-79987-8_8.
- 11 Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012. doi:10.1080/17445760.2012.668546.
- 12 Arnaud Casteigts, Anne-Sophie Himmel, Hendrik Molter, and Philipp Zschoche. Finding temporal paths under waiting time constraints. *Algorithmica*, 83(9):2754–2802, 2021. doi:10.1007/s00453-021-00831-w.
- 13 Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshantov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015. doi:10.1007/978-3-319-21275-3.
- 14 Reinhard Diestel. *Graph Theory*. Springer-Verlag, 2000.
- 15 Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999. doi:10.1007/978-1-4612-0515-9.
- 16 Thomas Erlebach, Michael Hoffmann, and Frank Kammer. On temporal graph exploration. *Journal of Computer and System Sciences*, 119:1–18, 2021. doi:10.1016/j.jcss.2021.01.005.
- 17 Thomas Erlebach and Jakob T. Spooner. Non-strict temporal exploration. In Andrea Werneck Richa and Christian Scheideler, editors, *27th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2020)*, volume 12156 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2020. doi:10.1007/978-3-030-54921-3_8.
- 18 Michael Held and Richard M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962. doi:10.1137/0110015.

- 19 David Kempe, Jon M. Kleinberg, and Amit Kumar. Connectivity and inference problems for temporal networks. *Journal of Computer and System Sciences*, 64(4):820–842, 2002. doi:10.1006/jcss.2002.1829.
- 20 George B. Mertzios, Hendrik Molter, Rolf Niedermeier, Viktor Zamaraev, and Philipp Zschoche. Computing maximum matchings in temporal graphs. In Christophe Paul and Markus Bläser, editors, *37th International Symposium on Theoretical Aspects of Computer Science (STACS 2020)*, volume 154 of *LIPICs*, pages 27:1–27:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.STACS.2020.27.
- 21 Othon Michail. An introduction to temporal graphs: An algorithmic perspective. *Internet Mathematics*, 12(4):239–280, 2016. doi:10.1080/15427951.2016.1177801.
- 22 Othon Michail and Paul G. Spirakis. Traveling salesman problems in temporal graphs. *Theoretical Computer Science*, 634:1–23, 2016. doi:10.1016/j.tcs.2016.04.006.
- 23 Hendrik Molter. *Classic graph problems made temporal – a parameterized complexity analysis*. PhD thesis, Technical University of Berlin, Germany, 2020. URL: <https://nbn-resolving.org/urn:nbn:de:101:1-2020120901012282017374>.
- 24 Moni Naor, Leonard J. Schulman, and Aravind Srinivasan. Splitters and near-optimal derandomization. In *IEEE 36th Annual Symposium on Foundations of Computer Science (FOCS 1995)*, pages 182–191. IEEE Computer Society, 1995. doi:10.1109/SFCS.1995.492475.
- 25 Herbert Robbins. A remark on Stirling’s formula. *The American Mathematical Monthly*, 62(1):26–29, 1955.
- 26 Claude E. Shannon. Presentation of a maze-solving machine. In Neil James Alexander Sloane and Aaron D. Wyner, editors, *Claude Elwood Shannon: Collected Papers*, pages 681–687. IEEE Press, 1993.
- 27 Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. Path problems in temporal graphs. *Proceedings of the VLDB Endowment*, 7(9):721–732, May 2014. doi:10.14778/2732939.2732945.
- 28 Philipp Zschoche, Till Fluschnik, Hendrik Molter, and Rolf Niedermeier. The complexity of finding small separators in temporal graphs. *Journal of Computer and System Sciences*, 107:72–92, 2020. doi:10.1016/j.jcss.2019.07.006.

Bipartite Temporal Graphs and the Parameterized Complexity of Multistage 2-Coloring

Till Fluschnik  

Algorithmics and Computational Complexity, Technische Universität Berlin, Germany

Pascal Kunz  

Algorithmics and Computational Complexity, Technische Universität Berlin, Germany

Abstract

We consider the algorithmic complexity of recognizing bipartite temporal graphs. Rather than defining these graphs solely by their underlying graph or individual layers, we define a bipartite temporal graph as one in which every layer can be 2-colored in a way that results in few changes between any two consecutive layers. This approach follows the framework of multistage problems that has received a growing amount of attention in recent years. We investigate the complexity of recognizing these graphs. We show that this problem is NP-hard even if there are only two layers or if only one change is allowed between consecutive layers. We consider the parameterized complexity of the problem with respect to several structural graph parameters, which we transfer from the static to the temporal setting in three different ways. Finally, we consider a version of the problem in which we only restrict the total number of changes throughout the lifetime of the graph. We show that this variant is fixed-parameter tractable with respect to the number of changes.

2012 ACM Subject Classification Theory of computation → Fixed parameter tractability; Theory of computation → Parameterized complexity and exact algorithms; Theory of computation → Dynamic graph algorithms

Keywords and phrases structural parameters, NP-hardness, parameterized algorithms, multistage problems

Digital Object Identifier 10.4230/LIPIcs.SAND.2022.16

Related Version *Full Version:* <https://arxiv.org/abs/2111.09049>

Funding *Till Fluschnik:* Supported by the DFG, project MATE (NI 369/17).

Pascal Kunz: Supported by the DFG Research Training Group 2434 “Facets of Complexity”.

1 Introduction

Bipartite graphs form a well-studied class of static graphs. A graph $G = (V, E)$ is bipartite if it admits a proper 2-coloring. A function $f: V \rightarrow \{1, 2\}$ is a *proper 2-coloring* of G if for all edges $\{v, w\} \in E$ it holds that $f(v) \neq f(w)$. In this work, we study the question of what a bipartite *temporal* graph is and how fast we can determine whether a temporal graph is bipartite. We approach this question through the prism of the novel program of multistage problems. Thus, we consider the following decision problem:

► Problem 1. MULTISTAGE 2-COLORING (MS2C)

Input: A temporal graph $\mathcal{G} = (V, (E_t)_{t=1}^\tau)$ and an integer $d \in \mathbb{N}_0$.

Question: Are there $f_1, \dots, f_\tau: V \rightarrow \{1, 2\}$ such that f_t is a proper 2-coloring for (V, E_t) for every $t \in \{1, \dots, \tau\}$ and $|\{v \in V \mid f_t(v) \neq f_{t+1}(v)\}| \leq d$ for every $t \in \{1, \dots, \tau - 1\}$?

In other words, (\mathcal{G}, d) is a **yes**-instance if \mathcal{G} admits a proper 2-coloring of each layer where only d vertices change colors between any two consecutive layers.



© Till Fluschnik and Pascal Kunz;

licensed under Creative Commons License CC-BY 4.0

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).

Editors: James Aspnes and Othon Michail; Article No. 16; pp. 16:1–16:18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

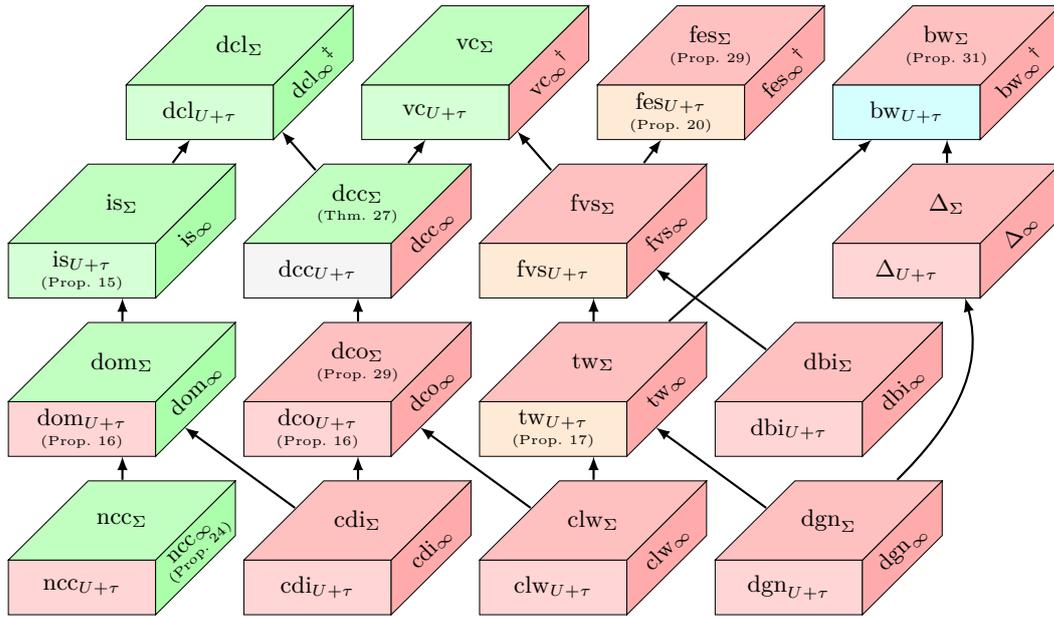
There have been various approaches to transferring graph classes from static to temporal graphs. If \mathcal{C} is a class of static graphs, then the two most obvious ways of defining a temporal analog to \mathcal{C} are (i) including all temporal graphs whose underlying graph is in \mathcal{C} or (ii) including all temporal graphs that have all of their layers in \mathcal{C} (see, for instance, [15]). Most applied research that has employed a notion of bipartiteness in temporal graphs [1, 24, 34] has defined it using the underlying graph, seeking to model relationships between two different types of entities. This is certainly appropriate as long as the type of an entity is not itself time-varying. Situations where entities can change their types require more sophisticated notions of bipartiteness. With MS2C, we model situations where we expect few entities to change their type between any two consecutive time steps. Later, in Section 5, we will consider a model for settings where we expect few changes overall.

The issue with both of the aforementioned classical approaches to defining temporal graph classes is that they do not take the time component into account when deciding membership in a class. For example, if the order of the layers is permuted arbitrarily, then this has no effect on membership in \mathcal{C} in either approach. Defining bipartiteness in the manner we propose does take the temporal order of the layers into consideration. It also leads to a hierarchy of temporal graph classes that are inclusion-wise between the two classes defined in the two aforementioned more traditional approaches: It is easy to see that $(\mathcal{G}, 0)$ is a **yes**-instance for MS2C if and only if the underlying graph of \mathcal{G} is bipartite. Conversely, if any layer of \mathcal{G} is not bipartite, then (\mathcal{G}, d) is a **no**-instance no matter the value of d . The two main drawbacks to defining temporal bipartiteness in this way are that (i) there is not one class of bipartite temporal graphs, but an infinite hierarchy depending on the value of d and (ii) as we will show, testing for bipartiteness in this sense is computationally much harder, but we will attempt to partially remedy this by analyzing the problem’s parameterized complexity for a variety of parameters.

Related work. The multistage framework is still young, but several problems have been investigated in it, mostly in the last couple of years, including MATCHING [3, 7, 18], KNAPSACK [4], s - t PATH [17], VERTEX COVER [16], COMMITTEE ELECTION [5], and others [2]. The framework has also been extended to goals other than minimizing the number of changes in the solution between layers [19, 22]. Since these types of problems are NP-hard even in fairly restricted settings, most research has focused on their parameterized complexity and approximability. MS2C is most closely related to MULTISTAGE 2-SAT [13] (see Section 2).

Our contributions. We prove that MS2C remains NP-hard even if $d = 1$ or if $\tau = 2$. We then analyze three ways of transferring structural graph parameters to the multistage setting: the maximum over the layers, the sum over all layers’ values, and its value on the underlying graph plus τ . We provide several (fixed-parameter) intractability and tractability results regarding these three notions of structural parameterizations (see Figure 1). Finally, we show that a slightly modified version of the problem in which there is no restriction on the number of changes between any two consecutive layers, but on the total number of changes throughout the lifetime of the graph, is fixed-parameter tractable with respect to the number of allowed changes. The proofs of statements marked with \star are deferred to the full version.

Discussion and outlook. While we proved that MS2C is NP-hard even if $\tau = 2$ or if $d = 1$, we leave open whether it is fixed-parameter tractable for the combined parameter $\tau + d$. We introduce a framework for analyzing the parameterized complexity of multistage problems regarding structural graph parameters. We resolve the parameterized complexity of MS2C



■ **Figure 1** Overview of selected structural parameters and our results (green: in FPT; orange: XP and W[1]-hard; red: para-NP-hard; blue: XP and open whether FPT or W[1]-hard; gray: open). [Δ : maximum degree; bw: bandwidth; cdi: diameter of connected component; clw: clique-width; dbi: distance to bipartite; dcc: distance to co-cluster; dcl: distance to clique; dco: distance to cograph; dgn: degeneracy; dom: domination number; fes: feedback edge number; fvs: feedback vertex number; is: independence number; ncc: number of connected components; tw: treewidth; vc: vertex cover number; for definitions of these parameters, see Section 2 or [32].] [†] (Proposition 25) [‡] (no polynomial kernel unless $\text{NP} \subseteq \text{coNP}$ / poly).

with respect to most of the parameters, but two cases are left open (cf. Figure 1). For instance, we proved that MS2C is in XP when parameterized by $\text{bw}_{U+\tau}$, but we do not know whether it is in FPT or W[1]-hard. Another interesting example is MS2C parameterized by $\text{dcc}_{U+\tau}$, for which we do not know whether it is contained in XP or para-NP-hard. Note that we proved fixed-parameter tractability regarding dcc_Σ . Finally, we suspect that it may also be worthwhile to investigate other multistage graph problem in our framework.

2 Preliminaries

We denote by \mathbb{N} (\mathbb{N}_0) the natural number excluding (including) zero. We use standard terminology from graph theory [9] and parameterized algorithmics [8].

Static and temporal graphs. We will frequently refer to graphs as static graphs in order to avoid confusion with temporal graphs. A static graph $G = (V, E)$ is *2-colorable* if it admits a proper 2-coloring. It is well-known that a static graph is 2-colorable if and only if contains no odd cycle. This can be checked in $\mathcal{O}(|V| + |E|)$ time by a simple search algorithm.

Let $G = (V, E)$ be a static graph. The *independence number* ($\text{is}(G)$) is the size of a largest set $X \subseteq V$ such that $G[X]$ is edgeless. The *domination number* ($\text{dom}(G)$) is the size of a smallest set $X \subseteq V$ such that every vertex in $V \setminus X$ has a neighbor in X . The maximum degree ($\Delta(G)$) is the maximum number of edges incident to a single vertex. A set of $X \subseteq V$ is a *connected component* if there is a path between any two vertices in X and no edge

16:4 Multistage 2-Coloring

between X and $V \setminus X$. We denote the *number of connected components* in G by $\text{ncc}(G)$. The *feedback edge number* ($\text{fes}(G)$) is $|E| - |V| + \text{ncc}(G)$ or equivalently the size of a minimum $X \subseteq E$ such that $G - X$ is acyclic. If S_n denotes the set of all permutations of $\{1, \dots, n\}$ and $V = \{v_1, \dots, v_n\}$, then the *bandwidth* ($\text{bw}(G)$) is $\min_{\pi \in S_n} \max_{\{v_i, v_j\} \in E} |\pi(i) - \pi(j)|$. A *tree decomposition* of G is a pair $(\mathcal{T}, \{\mathcal{X}_\alpha \mid \alpha \in V(\mathcal{T})\})$ where \mathcal{T} is a tree with node set $V(\mathcal{T})$ and $\mathcal{X}_\alpha \subseteq V$ for every $\alpha \in V(\mathcal{T})$ such that (i) $\bigcup_{\alpha \in V(\mathcal{T})} \mathcal{X}_\alpha = V$, (ii) for every $\{u, v\} \in E$ there is an $\alpha \in V(\mathcal{T})$ such that $u, v \in \mathcal{X}_\alpha$, and (iii) for every $v \in V$ the node set $\{\alpha \in V(\mathcal{T}) \mid v \in \mathcal{X}_\alpha\}$ induces a subtree of \mathcal{T} . The *width* of $(\mathcal{T}, \mathcal{X})$ is $\max_{\alpha \in V(\mathcal{T})} |\mathcal{X}_\alpha| - 1$. The *treewidth* ($\text{tw}(G)$) is the minimum width of a tree decomposition of G . If \mathcal{C} is a class of static graphs, then $X \subseteq V$ is a \mathcal{C} -*modulator* in G if $G - X \in \mathcal{C}$. The (i) *distance to cograph*, (ii) *vertex cover number*, (iii) *distance to bipartite*, and (iv) *distance to co-cluster* are the size of a minimum \mathcal{C} -modulator where \mathcal{C} is the set of all (i) cographs, (ii) edgeless graphs, (iii) bipartite graphs, and (iv) co-clusters, respectively.

A *temporal graph* $\mathcal{G} = (V, (E_t)_{t=1}^\tau)$ consists of a finite vertex set V and τ edge sets $E_1, \dots, E_\tau \subseteq \binom{V}{2}$. The *underlying graph* of \mathcal{G} is the static graph $\mathcal{G}_U := (V, \bigcup_{t=1}^\tau E_t)$. For $t \in \{1, \dots, \tau\}$, the t -th layer of \mathcal{G} is also a static graph, namely $\mathcal{G}_t := (V, E_t)$. The lifetime of \mathcal{G} is τ , the number of layers.

If $f_1, f_2: X \rightarrow Y$ are two functions that share a domain and a codomain, then $\delta(f_1, f_2) := |\{x \in X \mid f_1(x) \neq f_2(x)\}|$ is the number of elements of X whose value under f_1 differs from the value under f_2 .

Preliminary results. There is a connection between MS2C and the MULTISTAGE 2-SAT problem [13], which allows to transfer positive algorithmic results from the latter to the former.

► **Observation 1.** *There is a polynomial time algorithm that, taking an instance of MULTISTAGE 2-COLORING, constructs an equivalent instance of MULTISTAGE 2-SAT with n variables, $2m$ clauses, and $d' = d$.*

Proof. For each vertex v , construct a variable x_v . For each edge $\{v, w\}$ in a layer, construct the clauses $(x_v \vee x_w), (\overline{x_v} \vee \overline{x_w})$. ◀

Results on MULTISTAGE 2-SAT [13] hence imply the following.

► **Corollary 2.** MULTISTAGE 2-COLORING is (i) *polynomial-time solvable if $d \in \{0, n\}$* , (ii) *in XP regarding $n - d$ and $\tau + d$* , (iii) *FPT regarding $m + n - d$ and n* , and (iv) *admits a polynomial kernel regarding $m + \tau$ and $n + \tau$* .

We briefly note the following:

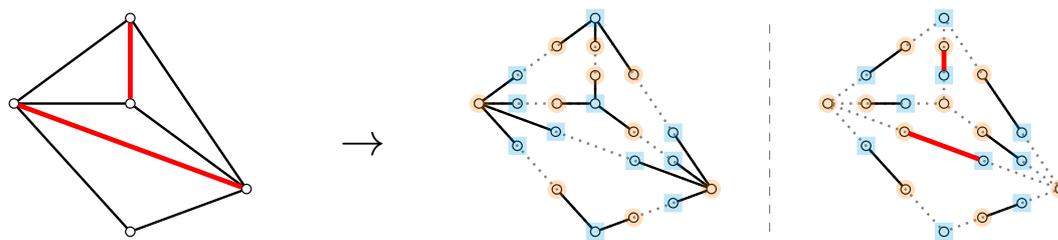
► **Observation 3.** *Given two 2-colorable graphs $G = (V, E)$ and $G' = (V, E')$, and two 2-colorings f of G and f' of G' , we can determine $\delta(f, f')$ in linear time.*

We can strengthen the first statement in Corollary 2 with the following proposition:

► **Proposition 4 (★).** MULTISTAGE 2-COLORING is *polynomial-time solvable if $d \geq \frac{1}{2}n$* .

Testing all sequences of functions $f_1, \dots, f_\tau: V \rightarrow \{1, 2\}$ gives us the following:

► **Observation 5.** MULTISTAGE 2-COLORING can be decided in time $\mathcal{O}(2^{\tau n} \cdot m)$ where τ is the lifetime, n the number of vertices, and m the number of time edges in a temporal graph.



■ **Figure 2** Illustration of Construction 1: The input graph G on the left hand-side (thick/red edges indicate a solution) and the output temporal graph \mathcal{G} on the right-hand side (thick/red edges in the second layer indicate where a recoloring was made; gray/dotted lines help to match with original edges from G).

3 NP-hard cases

We start by proving some complexity lower bounds for MULTISTAGE 2-COLORING. We will show that the problem is NP-hard in two fairly restricted cases.

3.1 Few changes allowed

► **Theorem 6 (★)**. MULTISTAGE 2-COLORING is NP-hard even for $d = 1$ and restricted to temporal graphs where each layer contains just three edges and has maximum degree one.

The proof is deferred to the full version.

3.2 Few stages

► **Theorem 7 (★)**. MULTISTAGE 2-COLORING is NP-hard on temporal graphs with at least two layers each of which is a forest.

To prove Theorem 7, we give a polynomial-time many-one reduction from the NP-complete [35] EDGE BIPARTIZATION problem defined by:

► **Problem 2.** EDGE BIPARTIZATION

Input: An undirected graph $G = (V, E)$ and $k \in \mathbb{N}_0$.

Question: Is there a set of edges $E' \subseteq E$ with $|E'| \leq k$ such that $G - E'$ is bipartite?

► **Construction 1.** Let $G = (V, E)$ be a graph and let $k \in \mathbb{N}_0$. We assume that $V = \{v_1, \dots, v_n\}$. We construct an instance (\mathcal{G}, d) of MS2C with $\mathcal{G} := (V', E_1, E_2)$ and $d := k$ as follows (see Figure 2 for an illustrative example).

The underlying graph of \mathcal{G} is obtained by subdividing each edge in G twice. Let u_i^e and u_j^e be the two vertices obtained by subdividing $e = \{v_i, v_j\}$ where u_i^e is adjacent to v_i and u_j^e to v_j . Then, $V' := V \cup \{u_i^e, u_j^e \mid e = \{v_i, v_j\} \in E\}$. The first layer of \mathcal{G} has edge set $E_1 := \{\{v_i, u_i^e\} \mid i \in \{1, \dots, n\}, e \in E, v_i \in e\}$. The second layer has edge set $E_2 := \{\{u_i^e, u_j^e\} \mid e = \{v_i, v_j\} \in E\}$. ▽

► **Lemma 8 (★)**. Instance (G, k) is a *yes-instance* for EDGE BIPARTIZATION if and only if instance (\mathcal{G}, d) output by Construction 1 is a *yes-instance* for MULTISTAGE 2-COLORING.

The reduction also implies the following:

► **Proposition 9.** *Unless the ETH fails, MULTISTAGE 2-COLORING admits no $\mathcal{O}(2^{o(n+m)})$ -time algorithm, where n is the number of vertices and m is the number of time edges in a temporal graph, even for $\tau = 2$.*

Proof. Unless the ETH fails, EDGE BIPARTIZATION cannot be solved in time $\mathcal{O}(2^{o(n)})$, where n is the number of vertices. This follows from the corresponding lower bound for MAXIMUM CUT [27]. The instance output by Construction 1 contains $n + 2m$ vertices. The claim follows by Lemma 8. ◀

4 Parameterized complexity

In the previous section we showed that MULTISTAGE 2-COLORING is NP-hard, even for constant values of τ and d . In this section, we study the parameterized complexity of MULTISTAGE 2-COLORING. To begin with, we will now show that MULTISTAGE 2-COLORING is fixed-parameter tractable with respect to $n - d$. This is in contrast to MULTISTAGE 2-SAT, which is W[1]-hard with respect to this parameter [13, Theorem 3.6].

► **Proposition 10.** *MULTISTAGE 2-COLORING is fixed-parameter tractable regarding $n - d$.*

Proof. If $d \geq \frac{n}{2}$, the problem can be solved in polynomial time (see Proposition 4). If $d < \frac{n}{2}$, then it follows that $n < 2(n - d)$. Hence, the fixed-parameter tractability of MS2C with respect to n (see Corollary 2) implies fixed-parameter tractability with respect to $n - d$. ◀

Additionally, we note the following kernelization lower bound.

► **Proposition 11 (★).** *Unless $\text{NP} \subseteq \text{coNP}/\text{poly}$, MULTISTAGE 2-COLORING admits no problem kernel of size polynomial in the number n of vertices.*

In the following, we will consider the parameterized complexity of MULTISTAGE 2-COLORING with respect to structural graph parameters. Research on the parameterized complexity of multistage problems has thus far mostly focused on the parameters that are given as part of the input such as d or τ . Although Fluschnik et al. [17] considered the vertex cover number and maximum degree of the underlying graph, there has been no systematic study of multistage problems concerned with structural parameters of the input temporal graph. We seek to initiate this line of research in the following. It follows the call by Fellows et al. [10, 12] to investigate problems’ “parameter ecology” in order to fully understand what makes them computationally hard. We will begin with a short discussion of how graph parameters can be applied to multistage problems. This question is closely related to issues that arise when applying such parameters to temporal graph problems (see [14] and [26, Sect. 2.4]).

A (temporal) graph parameter p is a function that maps any (temporal) graph G to a nonnegative integer $p(G)$. We will consider three ways of transferring graph parameters to temporal graphs. If p is a graph parameter, $\mathcal{G} = (V, (E_t)_{t=1}^\tau)$ is a temporal graph, $\mathcal{G}_t := (V, E_t)$ its t -th layer, and $\mathcal{G}_U := (V, \bigcup_{t=1}^\tau E_t)$ its underlying graph, then we define:

$$p_\infty(\mathcal{G}) := \max_{t \in \{1, \dots, \tau\}} p(\mathcal{G}_t), \quad (\text{maximum parameterization})$$

$$p_\Sigma(\mathcal{G}) := \sum_{t=1}^\tau \max\{1, p(\mathcal{G}_t)\}, \text{ and} \quad (\text{sum parameterization})$$

$$p_{U+\tau}(\mathcal{G}) := p(\mathcal{G}_U) + \tau. \quad (\text{underlying graph parameterization})$$

We will briefly explain our choice to define these parameters in this manner and describe the relationship between the parameters. For any two (temporal) graph parameters p_1 and p_2 , the first parameter p_1 is larger than p_2 , written $p_1 \succeq p_2$ or $p_2 \preceq p_1$, if there is a function $f: \mathbb{N}_0 \rightarrow \mathbb{N}_0$ such that $f(p_1(G)) \geq p_2(G)$ for all (temporal) graphs G . Such relationships between parameters are useful because, if $p_1 \succeq p_2$, then any problem that is fixed-parameter tractable with respect to p_2 is also fixed-parameter tractable with respect to p_1 . The \succeq -relation between static graph parameters is well-understood [21, 30, 31, 32, 33]. We will use these relationships implicitly and explicitly throughout this article. Many of the results claimed in Figure 1 will not be explicitly proved, because they are immediate consequences of other results and the \succeq -relation. The relationships under \succeq between selected graph parameters are pictured in that figure.

When it comes to transferring graph parameters from the static to the multistage setting, the parameters p_∞ and $p_{U+\tau}$ simply apply the graph parameter to the individual layers and to the underlying graph, respectively, and were used in a similar manner by Fluschnik et al. [14] and Molter [26]. The reasoning behind the definition of the sum parameterization may not be quite as obvious. It seems natural to consider the sum of the parameters over all layers. The issue with this is that it may not preserve the \succeq -relation. For example, it is well-known that feedback vertex number is a larger parameter (in the sense of \succeq) than treewidth. However, consider a temporal graph where each layer is a forest. Then, the sum of the feedback vertex numbers of the layers is 0, but the sum of the layers' treewidths is τ . Hence, treewidth is no longer bounded from above by the feedback vertex number. Our definition gets around this problem. In fact, all three aforementioned ways of transferring parameters from the static to the multistage setting preserve the \succeq -relation:

► **Proposition 12.** *Let p and q be graph parameters with $p \succeq q$. Then, $p_\alpha \succeq q_\alpha$ for any $\alpha \in \{\infty, \Sigma, U + \tau\}$.*

Proof. Let $f: \mathbb{N}_0 \rightarrow \mathbb{N}_0$ be a function such that $f(p(G)) \geq q(G)$ for all static graphs G . Without loss of generality, we may assume that (i) f is monotonically increasing, that is, $f(a) \geq f(b)$ if $a \geq b$, and (ii) $f(a) \geq a$ for every $a \in \mathbb{N}_0$ (consider $f'(a) := a + \max_{b \in \{1, \dots, a\}} f(b)$, $a \in \mathbb{N}_0$, for instance).

Let \mathcal{G} be an arbitrary temporal graph. Then:

$$f(p_\infty(\mathcal{G})) = f\left(\max_{t \in \{1, \dots, \tau\}} p(\mathcal{G}_t)\right) \stackrel{(i)}{=} \max_{t \in \{1, \dots, \tau\}} f(p(\mathcal{G}_t)) \geq \max_{t \in \{1, \dots, \tau\}} q(\mathcal{G}_t) = q_\infty(\mathcal{G})$$

For $n \in \mathbb{N}$, let $\text{Part}(n)$ denote the set of all partitions of n , that is all possible ways of writing n as $n = n_1 + n_2 + \dots + n_r$ for $r \geq 1$ and $n_i \in \mathbb{N}$. Let $g: \mathbb{N}_0 \rightarrow \mathbb{N}_0$ with:

$$g(0) := 0, \quad g(n) := \max \left\{ \sum_{i=1}^r f(n_i) \mid (n_1, \dots, n_r) \in \text{Part}(n) \right\} \text{ if } n > 0.$$

The maximum is well-defined, because $\text{Part}(n)$ is finite. Then, any temporal graph \mathcal{G} satisfies:

$$\begin{aligned} g(p_\Sigma(\mathcal{G})) &= g\left(\sum_{t=1}^{\tau} \max\{1, p(\mathcal{G}_t)\}\right) \geq \sum_{t=1}^{\tau} f(\max\{1, p(\mathcal{G}_t)\}) \stackrel{(i)}{=} \sum_{t=1}^{\tau} \max\{f(1), f(p(\mathcal{G}_t))\} \\ &\stackrel{(ii)}{\geq} \sum_{t=1}^{\tau} \max\{1, q(\mathcal{G}_t)\} = q_\Sigma(\mathcal{G}). \end{aligned}$$

(Note that the first inequality relies on the fact that every term in the sum is at least 1, since a partition can only be composed of positive summands. Therefore, this argument would not apply, if we defined the sum parameterization as simply the sum over the parameters of the individual layers.)

Lastly, for any temporal graph \mathcal{G} , we have:

$$g(p_{U+\tau}(\mathcal{G})) = g(p(\mathcal{G}_U) + \tau) \geq f(p(\mathcal{G}_u)) + f(\tau) \stackrel{(ii)}{\geq} q(\mathcal{G}_U) + \tau = q_{U+\tau}(\mathcal{G}). \quad \blacktriangleleft$$

Finally, we will briefly consider the relationship between p_∞ , p_Σ , and $p_{U+\tau}$. We will say that a graph parameter p is *monotonically increasing* if for any two static graphs $G = (V, E)$ and $G' = (V, E')$ with the same vertex set, it is the case that $E \subseteq E'$ implies $p(G) \leq p(G')$. Conversely, it is *monotonically decreasing* if $E \subseteq E'$ implies $p(G) \geq p(G')$.

► **Proposition 13.** *Let p be a graph parameter. Then:*

- (i) $p_\infty \preceq p_\Sigma$,
- (ii) $p_\Sigma \preceq p_{U+\tau}$, if p is monotonically increasing, and
- (iii) $p_\Sigma \succeq p_{U+\tau}$, if p is monotonically decreasing.

Proof.

(i) Obvious.

(ii) Let \mathcal{G} be a temporal graph. Note that since $\mathcal{G}_t \subseteq \mathcal{G}_U$, it follows that $p(\mathcal{G}_t) \leq p(\mathcal{G}_U)$ for all $t \in \{1, \dots, \tau\}$. Hence:

$$\begin{aligned} p_\Sigma(\mathcal{G}) &= \sum_{t=1}^{\tau} \max\{1, p(\mathcal{G}_t)\} \leq \tau + \sum_{t=1}^{\tau} p(\mathcal{G}_t) \leq \tau + \tau \cdot p(\mathcal{G}_U) \\ &\leq (\tau + p(\mathcal{G}_U))^2 = p_{U+\tau}(\mathcal{G})^2. \end{aligned}$$

(iii) Let \mathcal{G} be a temporal graph. Note that since $\mathcal{G}_t \subseteq \mathcal{G}_U$, it follows that $p(\mathcal{G}_t) \geq p(\mathcal{G}_U)$ for all $t \in \{1, \dots, \tau\}$. If $\tau = 1$ or $p(\mathcal{G}_U) \leq 1$, the claim is obvious. Otherwise, we have that:

$$\begin{aligned} p_\Sigma(\mathcal{G}) &= \sum_{t=1}^{\tau} \max\{1, p(\mathcal{G}_t)\} \geq \sum_{t=1}^{\tau} \max\{1, p(\mathcal{G}_U)\} \geq \sum_{t=1}^{\tau} p(\mathcal{G}_U) \\ &= \tau \cdot p(\mathcal{G}_U) \geq p_{U+\tau}(\mathcal{G}). \end{aligned} \quad \blacktriangleleft$$

We will now investigate the problem's parameterized complexity with respect to the three types of parameterizations. Figure 1 gives an overview of our results and of the abbreviations we use for the parameters. Our choice of parameters is partly motivated by Sorge and Weller's compendium [32] on graph parameters, but we limit our attention to those that are most interesting in the context of MS2C. For full definitions of the parameters, we refer the reader to Sorge and Weller's manuscript [32] or Section 2.

4.1 Underlying graph parameterization

► **Lemma 14.** *If $\mathcal{G} = (V, (E_t)_{t=1}^{\tau})$ is a temporal graph and every layer $\mathcal{G}_t = (V, E_t)$ of \mathcal{G} is bipartite for $t \in \{1, \dots, \tau\}$, then $\text{is}_{U+\tau}(\mathcal{G}) \geq 2^{-\tau}|V|$.*

Proof. (By induction on τ .) If $\tau = 1$, then \mathcal{G}_U is bipartite and the larger color class in any 2-coloring of \mathcal{G}_U forms an independent set containing at least $\frac{1}{2}|V|$ vertices. Suppose the claim holds for $\tau - 1$. Then, the underlying graph of $\mathcal{G}' = (V, (E_t)_{t=1}^{\tau-1})$ contains an independent set $X \subseteq V$ of size at least $2^{-(\tau-1)}|V|$. The graph $(X, \binom{X}{2} \cap E_\tau)$ is bipartite since it is a subgraph of (V, E_τ) . Hence, it contains an independent set Y of size at least $\frac{1}{2}|X| \geq 2^{-\tau}|V|$. Then, Y is also an independent set in \mathcal{G}_U . \blacktriangleleft

► **Proposition 15.** MULTISTAGE 2-COLORING is fixed-parameter tractable regarding $\text{is}_{U+\tau}$.

Proof. If any layer of \mathcal{G} is not bipartite, then the input can be immediately rejected. Otherwise, let \mathcal{G}_U be the underlying graph of \mathcal{G} . By Observation 5, MS2C can be solved in time $\mathcal{O}^*(2^{\tau \cdot |V|}) \leq \mathcal{O}^*(2^{\tau \cdot \text{is}_{U+\tau}(\mathcal{G}) \cdot 2^\tau})$. \blacktriangleleft

► **Proposition 16 (★).** MULTISTAGE 2-COLORING is NP-hard even if $\tau = 4$, $\text{dom}(\mathcal{G}_U) \leq 2$, and $\text{dco}(\mathcal{G}_U) = 0$. Hence, the problem is para-NP-hard with respect to $\text{dom}_{U+\tau}$ and $\text{dco}_{U+\tau}$.

► **Proposition 17 (★).** MULTISTAGE 2-COLORING can be solved in $\mathcal{O}^*(2^{\tau \cdot \text{tw}_{U+\tau}(\mathcal{G})} \cdot (d+1)^{2\tau})$ time. Hence, the problem is in XP when parameterized by $\text{tw}_{U+\tau}$.

The proof of this proposition utilizes a standard dynamic programming approach for problems parameterized by treewidth, extending it to the multistage context. Note that the running time of this algorithm also implies that MULTISTAGE 2-COLORING is fixed-parameter tractable with respect to $\tau + d + \text{tw}_{U+\tau}$.

► **Proposition 18 (★).** MULTISTAGE 2-COLORING is NP-hard even if $\tau = 3$ and $\Delta(G) = 3$. Hence, the problem is para-NP-hard with respect to $\Delta_{U+\tau}$.

► **Proposition 19 (★).** MULTISTAGE 2-COLORING is NP-hard even if $\tau = 3$ and $\text{dbi}_{U+\tau} = 2$.

Next, we will prove that MULTISTAGE 2-COLORING is W[1]-hard with respect to $\text{fes}_{U+\tau}$. In fact, we will prove the following slightly stronger statement:

► **Proposition 20 (★).** MULTISTAGE 2-COLORING is W[1]-hard when parameterized by τ , even if the feedback edge number $\text{fes}(\mathcal{G}_U)$ of the underlying graph is one.

We already showed that MS2C is XP regarding $\text{tw}_{U+\tau}$, so Proposition 20 implies that it is XP and W[1]-hard when parameterized by $\text{tw}_{U+\tau}$, $\text{fvs}_{U+\tau}$, and $\text{fes}_{U+\tau}$, since $\text{tw} \preceq \text{fvs} \preceq \text{fes}$. The proof of Proposition 20 is a little more involved than most of the previous hardness proofs. Our reduction is from the following:

► **Problem 3.** MULTICOLORED CLIQUE (MC)

Input: A k -colored static graph $G = (V, E)$ with $V = V_1 \uplus \dots \uplus V_k$.

Question: Does G contain a clique $X \subseteq V$ such that $|X \cap V_i| = 1$ for all $i \in \{1, \dots, k\}$?

MULTICOLORED CLIQUE is W[1]-hard when parameterized by k [11, 28].

► **Construction 2.** Let $(G = (V, E), k)$ with $V = V_1 \uplus \dots \uplus V_k$ be an instance of MULTICOLORED CLIQUE. We may assume that $|V_1| = \dots = |V_k| = n$ (if color classes do not have the same size, we can add isolated vertices), that all V_i are independent, and that $|E| \geq \binom{k}{2}$ (otherwise, this is clearly a no-instance). Let $V_i = \{v_0^i, \dots, v_{n-1}^i\}$.

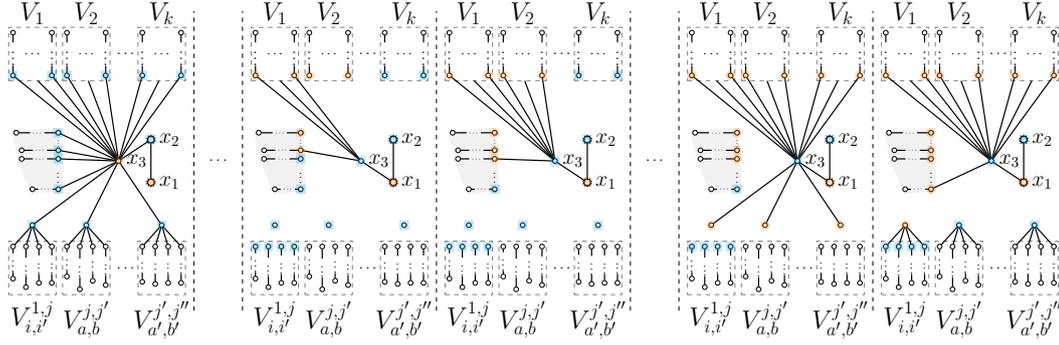
We will now describe an instance $(\mathcal{G} = (V', (E_t)_{t=1}^\tau), d)$ of MULTISTAGE 2-COLORING with $\text{fes}(\mathcal{G}_U) = 2$ (see Figure 3 for an illustration). We let $\tau := 2k(k-1) + 3$ and $d := |E|$.

The general idea behind the reduction is as follows. We consider the steps between consecutive layers and the number of changes to the coloring in those steps. The value of τ implies that there are $2k(k-1) + 2$ steps in total. There are $2k-2$ such steps for each color class in G , while the final two steps do not correspond to any color class. Of the $2k-2$ steps that correspond to $c \in \{1, \dots, k\}$, two will be used to verify adjacency to each of the $k-1$ other color classes. In order to be able to refer to these steps easily, we will use the following notation for any $c, c' \in \{1, \dots, k\}$, $c \neq c'$:

$$T(c \rightarrow c') := \begin{cases} 2(c-1)(k-1) + c', & \text{if } c > c', \\ 2(c-1)(k-1) + c' - 1, & \text{if } c < c', \end{cases} \text{ and}$$

$$T(c \Rightarrow c') := T(c \rightarrow c') + k - 1$$

16:10 Multistage 2-Coloring



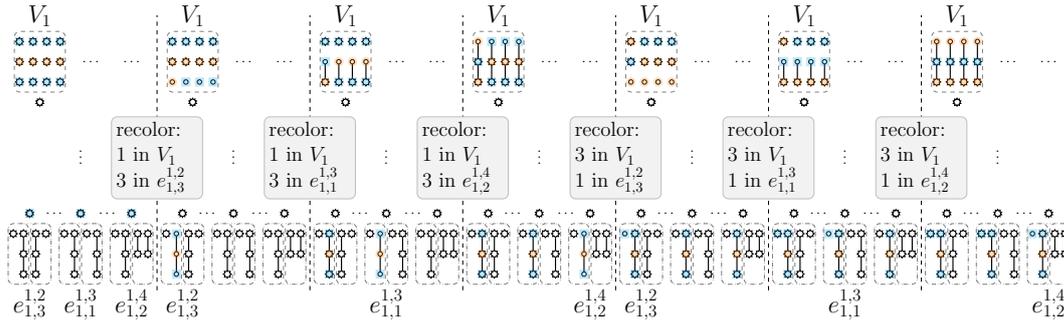
■ **Figure 3** Illustration of Construction 2. Shown are the first layer (left), the two layers when transitioning from the phase regarding V_2 to the phase regarding V_3 (middle), the last two layers (right). In the gray area, the waste-budget gadget is depicted. In this example, the edge $\{v_i^1, v_{i'}^j\}$ is chosen into the clique. Note that many vertices (those from paths and stars) are not depicted.

We will use several gadgets. The first gadget maintains its coloring throughout most of the lifetime of the instance. We use it to enforce a particular, predictable coloring on vertices in other gadgets at certain points. The second type of gadget represents the selection of a vertex in a certain color class. If the vertex v_j^i is to be added to the clique, it forces any multistage 2-coloring to make j changes in the first $k - 1$ steps corresponding to the color class i and $n - j - 1$ changes in the following $k - 1$ steps corresponding to this class. There is a third type of gadget. Its purpose is to verify that the vertices selected by the first gadget type are pairwise adjacent. There are numerous additional vertices whose sole purpose is to ensure that the coloring of vertices cannot change in unexpected ways. More specifically, when we say that a vertex v is *blocked* in time step t , we mean that we add d vertices that are adjacent to v in the $(t - 1)$ -st and t -th layers and isolated in all other layers. There are also further vertices designed to use up extraneous budget for changes during certain time steps.

We start by describing the first gadget, whose purpose is to maintain a predictable coloring so it can be used to enforce a certain coloring on other parts of the instance at particular points in time. This gadget contains the vertices x_1, x_2, x_3 . The edge $\{x_1, x_2\}$ is present in every layer of \mathcal{G} . The edge $\{x_2, x_3\}$ exists only in the first layer, while $\{x_1, x_3\}$ is present in all layers but the first. The vertices x_1 and x_2 are blocked in every step.

Next, we define the second type of gadget, which models the selection of a vertex in a color class. The gadget representing a certain color class V_c , $c \in \{1, \dots, k\}$, consists of $(n - 1)(k - 1)$ vertices $w_{i,j}^c$ for $i \in \{1, \dots, n - 1\}$, $j \in \{1, \dots, k - 1\}$. The vertex $w_{i,j}^c$ is blocked in all time steps except for the step $T(c \rightarrow j)$ and the step $T(c \Rightarrow j)$. There is an edge between $w_{i,j}^c$ and $w_{i,j+1}^c$ in the layers from $T(c \rightarrow j + 1)$ to $T(c \Rightarrow 1)$ and from $T(c \Rightarrow j)$ to $T(c + 1 \rightarrow 1)$. Additionally, in the very first and in the final layer of \mathcal{G} , all edges $\{w_{i,j}^c, w_{i,j+1}^c\}$ are present and there is an edge from x_3 to $w_{i,1}^c$ for all $c \in \{1, \dots, k\}$ and $i \in \{1, \dots, n - 1\}$. Moreover, for every $c \in \{1, \dots, k\}$, there is an edge from x_3 to $w_{i,1}^{c'}$ for all $i \in \{1, \dots, n - 1\}$ in all layers of index larger than $T(c \Rightarrow c')$, with $c' = \max\{1, \dots, k\} \setminus \{c\}$. This gadget is illustrated in the top part of Figure 4.

Next, we will describe the gadget that verifies that vertices selected in the previous gadget are pairwise adjacent. There is one such gadget for every edge $e = \{v_j^c, v_{j'}^{c'}\} \in E$, $1 \leq c < c' \leq k$, $j, j' \in \{0, \dots, n - 1\}$. The gadget consists of a root vertex u_0^e and four paths. The root is blocked in every step except for the final two. There is an edge between u_0^e and



■ **Figure 4** Illustrative example of the recolorings in Construction 2. Here, $n = 5, k = 4$, $e_{1,3}^{1,2} := \{v_1^1, v_3^2\}$, $e_{1,1}^{1,3} := \{v_1^1, v_1^3\}$, and $e_{1,2}^{1,4} := \{v_1^1, v_2^4\}$. The recolorings here represents the case that vertex v_1^1 is chosen into the clique, together with its incident edges to v_3^2, v_1^3 , and v_2^4 .

x_3 in the first and the $(\tau - 2)$ nd layer. The first vertex of each of the four paths is adjacent to u_0^e in the first and in the final layer. The edges of the paths are present in every layer. These paths consist of $n - 1 - j, j, n - 1 - j'$, and j' vertices, respectively. The vertices on the path of size $n - 1 - j$ are blocked in every time step except for step $T(c \rightarrow c')$. Those on the path of size j are blocked except for step $T(c \Rightarrow c')$. The vertices on the path of size $n - 1 - j'$ are blocked except for step $T(c' \rightarrow c)$. Finally, those on the path of size j' are blocked except for step $T(c' \Rightarrow c)$.

Finally, there is a gadget whose purpose is to waste extraneous budget for changes. It consists of $\tau - 2$ paths. There are $\tau - 4$ paths $P_3, \dots, P_{\tau-2}$ containing $d - (n - 1)$ vertices each, one path P_2 that consists of $d - n$ vertices, and one path P_τ that consists of $\binom{k}{2}$ vertices. For each $i \in \{2, \dots, \tau\} \setminus \{\tau - 1\}$, the first vertex in P_i is adjacent to x_3 exactly in the first and i th layer, where in all but the i th layer, all vertices from P_i are blocked. \square

The proof of the correctness of this reduction is deferred to the full version. We will briefly sketch a high-level description of this proof. All vertices in the gadget for a color class c must be re-colored at some point. Some number $i(k - 1)$ is re-colored in the first $k - 1$ steps corresponding to the color class and the remaining $(n - i - 1)(k - 1)$ are re-colored during the subsequent $k - 1$ steps (see Figure 4 for an illustration). That is, vertex v_i^c from color class c is added to the clique. In the final step, only $|E| - \binom{k}{2}$ vertices u_0^e can be re-colored. The other $\binom{k}{2}$ vertices correspond to edges that have both endpoints in the clique. The adjacency verification gadget ensures that, if u_0^e is not re-colored in the final step, then its endpoints must be selected to be part of the clique. This works because the four paths in this gadget must be re-colored in steps that belong to the color classes of the edge's endpoints.

► **Lemma 21 (★).** *The input instance to Construction 2 is a yes-instance for MULTICOLORED CLIQUE if and only if the output instance is a yes-instance for MULTISTAGE 2-COLORING.*

Finally, fixed-parameter tractability with respect to $vc_{U+\tau}$ can be proved using Theorem 27 (see Section 4.3) and the interplay between the different parameters (cf. Propositions 12 and 13).

► **Proposition 22 (★).** *MULTISTAGE 2-COLORING is fixed-parameter tractable regarding $vc_{U+\tau}$.*

4.2 Maximum parameterization

We turn our attention to the parameterized complexity of MULTISTAGE 2-COLORING with respect to several structural parameters under the maximum parameterization. We begin with ncc_∞ , the maximum number of connected components over all layers. Observe that under any proper 2-coloring the color of a single vertex determines the coloring of its entire connected component.

► **Observation 23.** *Every 2-colorable static graph with N connected components admits exactly 2^N different proper 2-colorings.*

This implies that MS2C is fixed-parameter tractable with respect to ncc_∞ .

► **Proposition 24.** MULTISTAGE 2-COLORING admits an $\mathcal{O}(4^{\text{ncc}_\infty(\mathcal{G})\tau})$ -time algorithm.

Proof. Let $N := \text{ncc}_\infty(\mathcal{G})$. We create an auxiliary static directed graph in the following manner. For each layer of \mathcal{G} , we include a node for every one of the at most 2^N many 2-colorings of this layer. There is a directed edge from a node representing a coloring of \mathcal{G}_t to a node representing a coloring of \mathcal{G}_{t+1} if the recoloring cost between the two is at most d . Finally, add two nodes s, t and connect s to every node corresponding to a coloring of the first layer and connect every node that corresponds to a coloring of the final layer to t . Then, (\mathcal{G}, d) is a **yes**-instance if and only if the auxiliary graph contains a path from s to t . Moreover, the auxiliary graph contains at most $\mathcal{O}(4^{\text{ncc}_\infty(\mathcal{G})\tau})$ edges. ◀

This result is essentially a stronger version of the statement in Corollary 2 that MULTISTAGE 2-COLORING is fixed-parameter tractable with respect to n , the number of vertices. However, ncc and larger parameters are the only structural parameters that yield fixed-parameter tractability with respect to the maximum parameterization.

► **Proposition 25 (★).** MULTISTAGE 2-COLORING is NP-hard even for constant values of vc_∞ , fes_∞ , and bw_∞ .

We note that Proposition 11 implies that MS2C does not admit a polynomial kernel for any parameter p listed in Figure 1 under the maximum parameterization, since $n \succeq p_\infty$ for all of these parameters.

4.3 Sum parameterization

We start with the parameterized complexity of MULTISTAGE 2-COLORING with respect to several structural parameters under the sum parameterization. For ncc_Σ , fixed-parameter tractability follows from that for ncc_∞ .

We start by proving that MS2C is fixed-parameter tractable with respect to the distance to co-cluster under the sum parameterization. This stands in contrast to the maximum parameterization (see Proposition 25). A graph is a co-cluster if and only if it does not contain $K_2 + K_1$ as an induced subgraph. By a general result obtained by Cai [6], this implies that the problem of determining whether $\text{dcc}(G) \leq k$ for a static graph G is fixed-parameter tractable with respect to k . We will make use of the following fact:

► **Observation 26.** *If G is a co-cluster, then G is edgeless or connected.*

► **Theorem 27.** MULTISTAGE 2-COLORING is fixed-parameter tractable regarding dcc_Σ .

If $G = (V, E)$ is a graph, then a function $\tilde{f}: V \rightarrow \{1, 2, \perp\}$ is a *proper partial 2-coloring* if the restriction of \tilde{f} to $V' := \{v \in V \mid \tilde{f}(v) \neq \perp\}$ is a proper 2-coloring of $G[V']$. If \tilde{f} is a proper partial and f is a proper 2-coloring of G , then f is an *extension* of \tilde{f} , if $\tilde{f}(v) \in \{\perp, f(v)\}$ for every $v \in V$. We will use the following as an intermediate problem.

► **Problem 4.** MULTISTAGE 2-COLORING EXTENSION (MS2CE)

Input: A temporal graph $\mathcal{G} = (V, (E_t)_{t=1}^\tau)$, proper partial 2-colorings $\tilde{f}_1, \dots, \tilde{f}_\tau: V \rightarrow \{1, 2, \perp\}$, and an integer $d \in \mathbb{N}_0$.

Question: Are there $f_1, \dots, f_\tau: V \rightarrow \{1, 2\}$ such that f_t is an extension of \tilde{f}_t and a proper 2-coloring of (V, E_t) for every $t \in \{1, \dots, \tau\}$ and $\delta(f_t, f_{t+1}) \leq d$ for every $t \in \{1, \dots, \tau - 1\}$?

We have the following immediate reduction rule for MS2CE.

► **Reduction Rule 1.** If an edge e has two colored endpoints, then delete e .

► **Lemma 28.** MULTISTAGE 2-COLORING EXTENSION is polynomial-time solvable if the input does not contain any edges.

Proof. We reduce MULTISTAGE 2-COLORING EXTENSION with no edges to the following job scheduling problem:

► **Problem 5.** $(1 \mid r_j, p_j = 1 \mid L_{\max})$ SCHEDULING

Input: A list of jobs j_1, \dots, j_n , where each job $j_i = (r_i, d_i)$ has a release date $r_i \in \mathbb{N}_0$ and a due date $d_i \in \mathbb{N}_0$, and a maximum lateness $L \in \mathbb{N}_0$.

Question: Is there a schedule $s: \{j_1, \dots, j_n\} \rightarrow \mathbb{N}_0$ such that (i) $s(j_i) \neq s(j_{i'})$ if $i \neq i'$, (ii) $s(j_i) \geq r_i$ for all $i \in \{1, \dots, n\}$, and (iii) $s(j_i) - d_i \leq L$ for all $i \in \{1, \dots, n\}$?

Horn [20, Sect. 2] showed that this scheduling problem can be solved by a polynomial-time greedy algorithm that always schedules the available job with the earliest due date. Let $(\mathcal{G} = (V, (\emptyset)_{t=1}^\tau), \tilde{f}_1, \dots, \tilde{f}_\tau, d)$ be an instance for MS2CE. We will say that vertex $v \in V$ between $t_1, t_2 \in \{1, \dots, \tau\}$ is *forced to be re-colored* $i \in \{1, 2\}$ if: (i) $t_1 < t_2$ and there is no t_3 with $t_1 < t_3 < t_2$ such that $\tilde{f}_{t_3}(v) \neq \perp$, (ii) $\tilde{f}_{t_2}(v) = i \in \{1, 2\}$, and (iii) $\tilde{f}_{t_1}(v) = 3 - i$. Let $\mathcal{R} \subseteq V \times \{1, \dots, \tau - 1\} \times \{2, \dots, \tau\} \times \{1, 2\}$ be the set of all forced re-colorings. Specifically, $(v, t_1, t_2, i) \in \mathcal{R}$ if and only if v is forced to be re-colored i between t_1 and t_2 .

In the machine scheduling model, only one job can be performed per time step, but, in a solution for an MS2C instance, up to d vertices can be re-colored. Hence, we fill each transition between two layers with d time slots. For $t \in \{1, \dots, \tau - 1\}$, the time slots $d(t - 1) + 1, \dots, dt$ correspond to changes in the coloring between the layers t and $t + 1$. For any forced re-coloring $(v, t_1, t_2, c) \in \mathcal{R}$, we create a job j_i with release date $r_i = d(t_1 - 1) + 1$ and due date $d_i = dt_2$. We will show that the given instance of MS2CE admits a solution if and only if this set of jobs admits a schedule with maximum lateness 0.

(\Rightarrow) Suppose that f_1, \dots, f_τ is a solution to the instance that extends $\tilde{f}_1, \dots, \tilde{f}_\tau$. It is easy to see that, if $(v, t_1, t_2, i) \in \mathcal{R}$, then $f_{t_1}(v) \neq f_{t_2}(v)$. Hence, there must be a t with $f_t(v) \neq f_{t+1}(v)$ and $t \in \{t_1, \dots, t_2 - 1\}$. Then, a machine schedule for the instance described above can be constructed by scheduling the job corresponding to (v, t_1, t_2, i) in one of the slots $d(t - 1) + 1, \dots, dt$. Since $\delta(f_t, f_{t+1}) \leq d$, there are enough slots.

(\Leftarrow) Suppose that we are given a machine schedule with maximum lateness 0 for the aforementioned instance. We construct an initial coloring f_1 by assigning each vertex v the color i , if there is a $t \in \{1, \dots, \tau\}$ such that $\tilde{f}_t(v) = i \in \{1, 2\}$ and $\tilde{f}_{t'}(v) = \perp$ for all $t' < t$. If $\tilde{f}_t(v) = \perp$ for all $t \in \{1, \dots, \tau\}$, then we assign $f_1(v)$ arbitrarily. We iteratively

■ **Algorithm 1** FPT-algorithm regarding dcc_Σ on input $\mathcal{G} = (V, (E_t)_{t=1}^\tau), d \in \mathbb{N}_0$.

```

1  $T^+, T^- \leftarrow \emptyset$ ;
2 foreach  $t \in \{1, \dots, \tau\}$  do
3    $X_t \leftarrow$  a minimum set such that  $G_t - X_t$  is a co-cluster;
4   if  $G_t - X_t$  is connected then  $T^+ \leftarrow T^+ \cup \{t\}$  else  $T^- \leftarrow T^- \cup \{t\}$ ;
5 foreach  $g_1: X_1 \rightarrow \{1, 2\}, \dots, g_\tau: X_\tau \rightarrow \{1, 2\}$  do //  $\leq 2^{\text{dcc}_\Sigma}$  many
6   foreach  $t \in \{1, \dots, \tau\}$  do
7     if  $t \in T^-$  then while  $\exists \{u, v\} \in E_t$  s.t.  $g_t(u) = i$  and  $g_t(v)$  is undefined, let
8        $g_t(v) \leftarrow 3 - i$ ;
9     if  $t \in T^+$  then  $F_t \leftarrow \{g_t^1, g_t^2\}$  with the two possible proper 2-colorings  $g_t^1, g_t^2$ 
10      of  $G_t - X_t$ ;
11   foreach  $(g_{t_1}^1, \dots, g_{t_{|T^-|}}^1) \in \times_{t \in T^-} F_t$  do //  $\leq 2^\tau$  many
12     Let  $\tilde{f}_t \leftarrow g_t$  if  $t \in T^-$  and  $\tilde{f}_t \leftarrow g_t \cup g_t^1$  if  $t \in T^+$ ;
13     if  $\tilde{f}_1, \dots, \tilde{f}_\tau$  are proper partial colorings then
14       if  $(\mathcal{G}, \tilde{f}_1, \dots, \tilde{f}_\tau, d)$  is a yes-instance for MS2CE then
15         return yes // decidable in polynomial time (Lemma 28)
16 return no

```

construct f_2, \dots, f_τ as follows. We let $f_{t+1}(v) = 3 - f_t(v)$ if the given schedule assigns a job j_i corresponding to a forced re-coloring $(v, t_1, t_2, 3 - f_t(v)) \in \mathcal{R}$ to a slot between $d(t-1) + 1$ and dt . Otherwise, we let $f_{t+1}(v) = f_t(v)$. ◀

The idea in the proof of Theorem 27 is as follows. After computing a distance-to-co-cluster set for each layer, we check for all possible colorings of these sets, and then propagate the colorings. We finally arrive at an instance of MS2CE with no edges, which is decidable in polynomial time.

Proof of Theorem 27. Let $\mathcal{I} = (\mathcal{G}, d)$ be an instance of MULTISTAGE 2-COLORING. Let $\mathcal{G} = (V, (E_t)_{t=1}^\tau)$ and $G_t := (V, E_t)$ be the t -th layer of G . Let $k := \sum_{t=1}^\tau \text{dcc}(G_t)$. The following algorithm is summarized in pseudocode in Algorithm 1.

For each $t \in \{1, \dots, \tau\}$, using Cai's algorithm [6], we can compute in $2^{\mathcal{O}(k)} \cdot |G_t|^{\mathcal{O}(1)}$ time a minimum set $X_t \subseteq V$ such that $G_t - X_t$ is a co-cluster. Let (T^+, T^-) be a partition of $\{1, \dots, \tau\}$ such that $t \in T^+$ if and only if $G_t - X_t$ is connected (see Observation 26). For $t \in T^+$, let $V_t := V(G_t - X_t)$, and for $t \in T^-$, let $V_t := \{v \in V(G_t - X_t) \mid \deg_{G_t}(v) > 0\}$ be the vertices in $G_t - X_t$ incident to at least one edge in G_t . We then iterate over all the at most 2^k possible partial 2-colorings of (X_1, \dots, X_τ) . For every $t \in T^+$ there are only two possible proper 2-colorings of $G_t - X_t$. We iterate over all the at most 2^τ possible 2-colorings of these layers. For every $t \in T^-$, if there is an uncolored vertex v with a neighbor w colored $i \in \{1, 2\}$, then color v with color $3 - i$. Note that this colors all vertices in V_t . Let $\tilde{f}_1, \dots, \tilde{f}_\tau$ be the resulting partial coloring. The important thing to note is that for every $t \in \{1, \dots, \tau\}$ and every edge in E_t both its endpoints are colored by \tilde{f}_t . If one of $\tilde{f}_1, \dots, \tilde{f}_\tau$ is not proper, we reject the coloring, otherwise we proceed as follows.

Construct the instance $\tilde{\mathcal{I}} = (\mathcal{G}, (\tilde{f}_t)_{t=1}^\tau, d)$ of MULTISTAGE 2-COLORING EXTENSION. Since every edge has two colored endpoints, applying Reduction Rule 1 exhaustively results in an instance $\tilde{\mathcal{I}}' = (\mathcal{G}', (\tilde{f}_t)_{t=1}^\tau, d)$ of MULTISTAGE 2-COLORING EXTENSION where \mathcal{G}' contains no edge. Hence, due to Lemma 28, we can solve $\tilde{\mathcal{I}}'$ in polynomial-time. Thus, the overall running time is in $\sum_{t=1}^\tau 2^{\mathcal{O}(k)} \cdot |G_t|^{\mathcal{O}(1)} + 2^{k+\tau} |\mathcal{G}|^{\mathcal{O}(1)}$.

Clearly, if $\tilde{\mathcal{I}}'$ is a **yes**-instance in one choice, then \mathcal{I} is a **yes**-instance of MS2C. That the opposite direction is correct too is also not hard to see. Note that every solution f_1, \dots, f_τ induces a proper partial coloring $\tilde{f}_1, \dots, \tilde{f}_\tau$, where \tilde{f}_t is induced on $V_t \cup X_t$ for every $t \in \{1, \dots, \tau\}$, that we will eventually check. Moreover, the resulting input to MS2CE is clearly a **yes**-instance: f_1, \dots, f_τ is a solution to $(\mathcal{G}, (\tilde{f}_t)_{t=1}^\tau, d)$. ◀

► **Proposition 29 (★)**. MULTISTAGE 2-COLORING is NP-hard even for constant values of (i) dco_Σ , (ii) fes_Σ , and (iii) Δ_Σ .

Our final result on structural parameters concerns bw_Σ , that is, bandwidth with the sum parameterization. We first briefly note the following:

► **Observation 30**. Let G be an undirected graph. If every connected component in G contains at most k vertices, then $\text{bw}(G) \leq k - 1$.

We can use this observation to show that MULTISTAGE 2-COLORING is para-NP-hard when parameterized by bw_Σ .

► **Proposition 31 (★)**. MULTISTAGE 2-COLORING is NP-hard even if bw_Σ is constant.

5 Global budget

The problem we have considered so far is the multistage version of 2-COLORING with a *local* budget. Heeger et al. [19] started the parameterized research of multistage graph problems on a *global* budget where there is no restriction on the number of changes between any two consecutive layers, but instead a restriction on the total number of changes made throughout the lifetime of the instance. All graph problems studied by Heeger et al. are NP-hard even for constant values of the global budget parameter. By contrast, we will show that a global budget version of MULTISTAGE 2-COLORING is fixed-parameter tractable with respect to the budget. Formally, the global budget version of MULTISTAGE 2-COLORING is:

► **Problem 6**. MULTISTAGE 2-COLORING ON A GLOBAL BUDGET (MS2CGB)

Input: A temporal graph $\mathcal{G} = (V, (E_t)_{t=1}^\tau)$ and an integer $D \in \mathbb{N}_0$.

Question: Are there $f_1, \dots, f_\tau: V \rightarrow \{1, 2\}$ such that f_t is a proper 2-coloring of (V, E_t) for every $t \in \{1, \dots, \tau\}$ and $\sum_{t=1}^{\tau-1} \delta(f_t, f_{t+1}) \leq D$?

We start by pointing out that MS2CGB is NP-hard. This follows from Theorem 7, since there is no distinction between a local and a global budget if $\tau = 2$.

► **Observation 32**. MULTISTAGE 2-COLORING ON A GLOBAL BUDGET is NP-hard.

In order to show that MULTISTAGE 2-COLORING ON A GLOBAL BUDGET is fixed-parameter tractable, we will prove the existence of a parameter-preserving transformation to the ALMOST 2-SAT problem, which is defined by:

► **Problem 7**. ALMOST 2-SAT (A2SAT)

Input: A Boolean formula φ in 2-CNF and an integer k .

Question: Can φ be made satisfiable by removing at most k clauses?

Razgon and O'Sullivan [29] prove that A2SAT is fixed-parameter tractable when parameterized by k , but the fastest presently known algorithm runs in $\mathcal{O}^*(2.3146^k)$ and is due to Lokshtanov et al. [25]. Kratsch and Wahlström [23] show that this problem admits a randomized polynomial kernel.

► **Proposition 33 (★).** MULTISTAGE 2-COLORING ON A GLOBAL BUDGET *parameterized by D* admits a *parameter-preserving transformation to ALMOST 2-SAT parameterized by k* .

The proof is deferred to the full version. The basic idea behind the reduction is that we use $D + 1$ copies of the same two clauses to express that no edge should be monochromatic. At least one of these clause pairs must survive the deletion. Moreover, we add clauses stating that vertices are not re-colored. At most D of these clauses can be deleted. This directly implies the following:

► **Corollary 34.** MULTISTAGE 2-COLORING ON A GLOBAL BUDGET *parameterized by D* is *fixed-parameter tractable and admits a randomized polynomial kernel*.

We note that the approach described here for MS2C can be used to reduce a global budget version of the more general MULTISTAGE 2-SAT to ALMOST 2-SAT, proving the following:

► **Observation 35.** MULTISTAGE 2-SAT ON A GLOBAL BUDGET *parameterized by the number of changes* is *fixed-parameter tractable and admits a randomized polynomial kernel*.

References

- 1 Shorouq Al-Eidi, Yuanzhu Chen, Omar Darwishand, and Ali M. S. Alfosoool. Time-ordered bipartite graph for spatio-temporal social network analysis. In *Proceedings of the 2020 International Conference on Computing, Networking and Communications (ICNC)*, pages 833–838, 2020. doi:10.1109/ICNC47757.2020.9049668.
- 2 Evripidis Bampis, Bruno Escoffier, and Alexander V. Kononov. LP-based algorithms for multi-stage minimization problems. In *Proceedings of the 18th International Workshop on Approximation and Online Algorithms (WAOA)*, pages 1–15, 2020. doi:10.1007/978-3-030-80879-2_1.
- 3 Evripidis Bampis, Bruno Escoffier, Michael Lampis, and Vangelis Th. Paschos. Multistage matchings. In *Proceedings of the 16th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT)*, pages 7:1–7:13, 2018. doi:10.4230/LIPIcs.SWAT.2018.7.
- 4 Evripidis Bampis, Bruno Escoffier, and Alexandre Teiller. Multistage knapsack. In *Proceedings of the 44th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 22:1–22:14, 2019. doi:10.4230/LIPIcs.MFCS.2019.22.
- 5 Robert Bredereck, Till Fluschnik, and Andrzej Kaczmarczyk. Multistage committee election. *arXiv*, 2020. arXiv:2005.02300.
- 6 Leizhen Cai. Fixed-parameter tractability of graph modification problems for hereditary properties. *Information Processing Letters*, 58(4):171–176, 1996. doi:10.1016/0020-0190(96)00050-6.
- 7 Markus Chimani, Niklas Troost, and Tilo Wiedera. Approximating multistage matching problems. In *Proceedings of the 32nd International Workshop on Combinatorial Algorithms (IWOCA)*, pages 558–570, 2021. doi:10.1007/978-3-030-79987-8_39.
- 8 Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshantov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015. doi:10.1007/978-3-319-21275-3.
- 9 Reinhard Diestel. *Graph Theory*. Springer, 5th edition, 2017. doi:10.1007/978-3-662-53622-3.
- 10 Michael Fellows, Daniel Lokshantov, Neeldhara Misra, Matthias Mnich, Frances Rosamond, and Saket Saurabh. The complexity ecology of parameters: An illustration using bounded max leaf number. *Theory of Computing Systems*, 45(4):822–848, 2009. doi:10.1007/s00224-009-9167-9.

- 11 Michael R. Fellows, Danny Hermelin, Frances A. Rosamond, and Stéphane Vialette. On the parameterized complexity of multiple-interval graph problems. *Theoretical Computer Science*, 410(1):53–61, 2009. doi:10.1016/j.tcs.2008.09.065.
- 12 Michael R. Fellows, Bart M.P. Jansen, and Frances Rosamond. Towards fully multivariate algorithmics: Parameter ecology and the deconstruction of computational complexity. *European Journal of Combinatorics*, 34(3):541–566, 2013. doi:10.1016/j.ejc.2012.04.008.
- 13 Till Fluschnik. A multistage view on 2-satisfiability. In *Proceedings of the 12th International Conference on Algorithms and Complexity (CIAC)*, pages 231–244, 2021. doi:10.1007/978-3-030-75242-2_16.
- 14 Till Fluschnik, Hendrik Molter, Rolf Niedermeier, Malte Renken, and Philipp Zschoche. As time goes by: Reflections on treewidth for temporal graphs. In Fedor V. Fomin, Stefan Kratsch, and Erik Jan van Leeuwen, editors, *Treewidth, Kernels, and Algorithms: Essays Dedicated to Hans L. Bodlaender on the Occasion of His 60th Birthday*, pages 49–77. Springer International Publishing, Cham, 2020. doi:10.1007/978-3-030-42071-0_6.
- 15 Till Fluschnik, Hendrik Molter, Rolf Niedermeier, Malte Renken, and Philipp Zschoche. Temporal graph classes: A view through temporal separators. *Theoretical Computer Science*, 806:197–218, 2020. doi:10.1016/j.tcs.2019.03.031.
- 16 Till Fluschnik, Rolf Niedermeier, Valentin Rohm, and Philipp Zschoche. Multistage vertex cover. In *Proceedings of the 14th International Symposium on Parameterized and Exact Computation (IPEC)*, pages 14:1–14:14, 2019. doi:10.4230/LIPIcs.IPEC.2019.14.
- 17 Till Fluschnik, Rolf Niedermeier, Carsten Schubert, and Philipp Zschoche. Multistage s -path: Confronting similarity with dissimilarity in temporal graphs. In *Proceedings of the 31st International Symposium on Algorithms and Computation (ISAAC)*, pages 43:1–43:16, 2020. doi:10.4230/LIPIcs.ISAAC.2020.43.
- 18 Anupam Gupta, Kunal Talwar, and Udi Wieder. Changing bases: Multistage optimization for matroids and matchings. In *Proceedings of the 41st International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 563–575, 2014. doi:10.1007/978-3-662-43948-7_47.
- 19 Klaus Heeger, Anne-Sophie Himmel, Frank Kammer, Rolf Niedermeier, Malte Renken, and Andrej Sajenko. Multistage graph problems on a global budget. *Theoretical Computer Science*, 868:46–64, 2021. doi:10.1016/j.tcs.2021.04.002.
- 20 W. A. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21(1):177–185, 1974. doi:10.1002/nav.3800210113.
- 21 Bart M. P. Jansen. *The Power of Data Reduction: Kernels for Fundamental Graph Problems*. PhD thesis, Utrecht University, 2013. URL: <http://dspace.library.uu.nl/handle/1874/276438>.
- 22 Leon Kellerhals, Malte Renken, and Philipp Zschoche. Parameterized algorithms for diverse multistage problems. In *Proceedings of the 29th Annual European Symposium on Algorithms (ESA)*, pages 55:1–55:17, 2021. doi:10.4230/LIPIcs.ESA.2021.55.
- 23 Stefan Kratsch and Magnus Wahlström. Representative sets and irrelevant vertices: New tools for kernelization. *Journal of the ACM*, 67(3), June 2020. doi:10.1145/3390887.
- 24 Matthieu Latapy, Clémence Magnien, and Tiphaine Viard. Weighted, bipartite, or directed stream graphs for the modeling of temporal networks. In Petter Holme and Jari Saramäki, editors, *Temporal Network Theory*, pages 49–64. Springer International Publishing, Cham, 2019. doi:10.1007/978-3-030-23495-9_3.
- 25 Daniel Lokshtanov, N. S. Narayanaswamy, Venkatesh Raman, M. S. Ramanujan, and Saket Saurabh. Faster parameterized algorithms using linear programming. *ACM Transactions on Algorithms*, 11(2):1–31, 2014. doi:10.1145/2566616.

- 26 Hendrik Molter. *Classic Graph Problems Made Temporal: A Parameterized Complexity Analysis*. PhD thesis, Technische Universität Berlin, 2020. doi:10.14279/depositonce-10551.
- 27 Karolina Okrasa and Paweł Rzażewski. Subexponential algorithms for variants of the homomorphism problem in string graphs. *Journal of Computer and System Sciences*, 109:126–144, 2020. doi:10.1016/j.jcss.2019.12.004.
- 28 Krzysztof Pietrzak. On the parameterized complexity of the fixed alphabet shortest common supersequence and longest common subsequence problems. *Journal of Computer and System Sciences*, 67(4):757–771, 2003. doi:10.1016/S0022-0000(03)00078-3.
- 29 Igor Razgon and Barry O’Sullivan. Almost 2-SAT is fixed-parameter tractable. *Journal of Computer and System Sciences*, 75(8):435–450, 2009. doi:10.1016/j.jcss.2009.04.002.
- 30 Róbert Sasák. Comparing 17 graph parameters. Master’s thesis, University of Bergen, 2010. URL: <https://bora.uib.no/bora-xmlui/handle/1956/4329>.
- 31 Johannes Schröder. Comparing graph parameters. Bachelor’s thesis, Technische Universität Berlin, 2019. URL: <http://fpt.akt.tu-berlin.de/publications/theses/BA-Schröder.pdf>.
- 32 Manuel Sorge and Mathias Weller. The graph parameter hierarchy. Unpublished manuscript, 2019. URL: <https://manyu.pro/assets/parameter-hierarchy.pdf>.
- 33 Martin Vatshelle. *New Width Parameters of Graphs*. PhD thesis, University of Bergen, 2012. URL: <https://www.ii.uib.no/~martinv/Papers/MartinThesis.pdf>.
- 34 Tsunghan Wu, Sheau-Harn Yu, Wanjiun Liao, and Cheng-Shang Chang. Temporal bipartite projection and link prediction for online social networks. In *Proceedings of the 2014 IEEE International Conference on Big Data (Big Data)*, pages 52–59, 2014. doi:10.1109/BigData.2014.7004444.
- 35 Mihalis Yannakakis. Node- and edge-deletion NP-complete problems. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing (STOC)*, pages 253–264, 1978. doi:10.1145/800133.804355.

Temporal Connectivity: Coping with Foreseen and Unforeseen Delays

Eugen Füchsle ✉

Faculty IV, Algorithmics and Computational Complexity, TU Berlin, Germany

Hendrik Molter ✉ 

Department of Industrial Engineering and Management,
Ben-Gurion University of the Negev, Beer-Sheva, Israel

Rolf Niedermeier ✉ 

Faculty IV, Algorithmics and Computational Complexity, TU Berlin, Germany

Malte Renken ✉ 

Faculty IV, Algorithmics and Computational Complexity, TU Berlin, Germany

Abstract

Consider planning a trip in a train network. In contrast to, say, a road network, the edges are *temporal*, i.e., they are only available at certain times. Another important difficulty is that trains, unfortunately, sometimes get delayed. This is especially bad if it causes one to miss subsequent trains. The best way to prepare against this is to have a connection that is *robust* to some number of (small) delays. An important factor in determining the robustness of a connection is how far in advance delays are announced. We give polynomial-time algorithms for the two extreme cases: delays known before departure and delays occurring without prior warning (the latter leading to a two-player game scenario). Interestingly, in the latter case, we show that the problem becomes PSPACE-complete if the itinerary is demanded to be a path.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Theory of computation → Problems, reductions and completeness; Mathematics of computing → Discrete mathematics

Keywords and phrases Paths and walks in temporal graphs, static expansions of temporal graphs, two-player games, flow computations, dynamic programming, PSPACE-completeness

Digital Object Identifier 10.4230/LIPIcs.SAND.2022.17

Related Version *Full Version*: <https://arxiv.org/abs/2201.05011> [15]

Funding *Hendrik Molter*: Supported by the ISF, grant No. 1070/20.

Malte Renken: Supported by the DFG, project MATE (NI 369/17).

1 Introduction

Computing *temporal paths* is one of the back-bone algorithmic problems in the context of temporal graphs, that is, graphs whose edges are present only at certain, known points in time [3, 26]. Temporal graphs are specified by a set V of vertices and a set E of time arcs, where each time arc $(v, w, t, \lambda) \in E$ consists of a *start vertex* v , an *end vertex* w , a *time label* t , and a *traversal time* λ ; then there is a (direct) connection from v to w starting at time t and arriving at time $t + \lambda$. Temporal graphs model numerous real-world scenarios [16, 17, 5, 20]: Social, communication, transportation, and many other networks are usually not static but vary over time.

The added dimension of time causes many aspects of connectivity to behave quite differently from static (i.e., non-temporal) graphs. Thus, the flow of items through a temporal network has to be time-respecting. More specifically, it follows a *temporal walk* (or *path*, if every vertex is visited at most once), i.e., a sequence of time arcs $(v_i, w_i, t_i, \lambda_i)_{i=1}^{\ell}$



© Eugen Füchsle, Hendrik Molter, Rolf Niedermeier, and Malte Renken;
licensed under Creative Commons License CC-BY 4.0

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).

Editors: James Aspnes and Othon Michail; Article No. 17; pp. 17:1–17:17

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

where $v_{i+1} = w_i$ and $t_{i+1} \geq t_i + \lambda_i$ for all $i < \ell$. While inheriting many properties from static walks, temporal walks exhibit certain characteristics that add a further level of difficulty to algorithmic problems centered around them. For example, temporal connectivity is not transitive: the existence of a temporal walk from vertex u to v and a temporal walk from v to w does not imply the existence of a temporal walk from u to w . Moreover, the temporal setting naturally leads to several notions of what an “optimal” temporal path could be [2, 3].

Computation of temporal paths and walks has already been studied intensively [26, 3], including specialized settings that are novel to temporal graphs: For example, Bentert et al. [2] and Casteigts et al. [6] studied temporal walks and paths that are only allowed to have limited waiting time at any vertex.

In this work, we investigate another natural, inherently temporal connectivity problem. It addresses *delays*. In many real-world temporal networks such as transport networks (e.g. trains, shipping routes), individual edges may get delayed for various reasons. Thus it is an important question whether connectivity between a start and a target node is *fragile*, i.e., easily disrupted by delays, or whether it is *robust*.

An important aspect in this matter is the time at which delays become known. The earlier they are announced, the easier one can still adapt the chosen route. In this work, we study the two endpoints of this spectrum: one where all delays are known up front, and one where delays occur without any prior warning.

We now briefly describe our models for these two problems, beginning with the problem variant in which all delays are announced before the start of the journey, and the question is whether the designated target remains reachable even in a worst-case scenario. Herein, a *D-delayed temporal path* refers to a temporal path that remains valid when the time arcs in $D \subseteq E$ have been delayed by some fixed amount δ each. A more formal definition will be given in Section 2.

DELAY-ROBUST CONNECTION

Input: A temporal graph $\mathcal{G} = (V, E)$, two vertices $s, z \in V$, and $x, \delta \in \mathbb{N}$.

Question: Is there, for every delay set $D \subseteq E$ of size $|D| \leq x$, a D -delayed temporal path from s to z in \mathcal{G} ?

In contrast, if the delays occur during the journey without prior warning, then the resulting problem is best modeled as a two-player game, which we call the Delayed-Routing Game. The first player (the *traveler*) starts at vertex s and has to decide at each turn which time arc they want to traverse next. The other player (the *adversary*) then gets to decide whether that time arc is delayed or not. As before, there is a bound x on the overall number of time arcs that can be delayed. The traveler wins if they reach the target vertex z . A *winning* strategy for the traveler is a strategy that guarantees that they will reach their target.

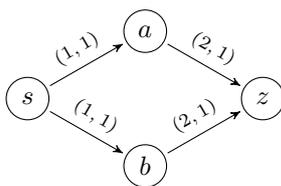
DELAYED-ROUTING GAME

Input: A temporal graph $\mathcal{G} = (V, E)$, two vertices $s, z \in V$, and $x, \delta \in \mathbb{N}$.

Question: Does the traveler have a winning strategy for the Delayed-Routing Game?

The difference between the two models DELAY-ROBUST CONNECTION and DELAYED-ROUTING GAME is illustrated in Figure 1.

Finally, we want to consider a variant of DELAYED-ROUTING GAME, in which the traveler may not visit any vertex more than once. We refer to this as DELAYED-ROUTING PATH GAME.



■ **Figure 1** An example temporal graph, time arcs are labeled by $(t(e), \lambda(e))$. For $x = 1$ and $\delta = 1$, this is a yes-instance of DELAY-ROBUST CONNECTION: If the time arc from s to a is delayed then there is a temporal path from s to z via b . Otherwise, if that time arc is not delayed, then the temporal path via a is available. However, the same setting is a no-instance of DELAYED-ROUTING GAME: If the traveler picks the time arc from s to a , then the adversary will delay it, rendering the traveler stuck at a since they reach it at time 3. If the traveler picks the time arc from s to b , then the situation is analogous.

Related Work. There has been extensive research on many other connectivity-related problems on temporal graphs [4, 11, 19, 21, 9, 10, 12, 27, 18]. Delays in temporal graphs have been considered in terms of manipulating reachability sets [7, 22]. An individual delay operation considered in the mentioned work delays a single time arc and is similar to our notion. Typically, the computational problems in this context are NP-hard and can be also considered as computing “robustness measures” for the connectivity in temporal graphs.

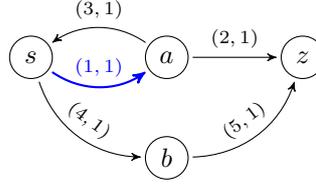
In companion work [14] we investigate a problem located somewhat “between” DELAY-ROBUST CONNECTION and DELAYED-ROUTING GAME in which the delays become known after the sequence of vertices to be traversed from s to z is fixed, but before the exact time arcs to be traversed are chosen. There, we show that this problem is NP-hard and further study its parameterized complexity.

Our Contribution. We introduce two computational problems related to testing connectivity between two terminal vertices in the presence of delays. We give polynomial-time algorithms for DELAY-ROBUST CONNECTION (Section 3) and DELAYED-ROUTING GAME (Section 4.1), but prove PSPACE-completeness for DELAYED-ROUTING PATH GAME (Sections 4.2 and 4.3), the variant of the second problem in which no vertex may be visited twice. Due to space constraints, the PSPACE-hardness proof for DELAYED-ROUTING PATH GAME is partially deferred to a full version [15].

2 Preliminaries

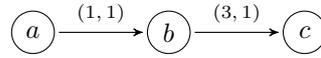
We abbreviate $\{1, 2, \dots, n\}$ as $[n]$ and $\{n, n + 1, \dots, m\}$ as $[n, m]$. The Iverson bracket $[P]$ is 1 if property P holds and 0 otherwise. For a time arc $e = (v, w, t, \lambda_e)$, we denote the starting and ending vertices as $\text{start}(e) = v$ and $\text{end}(e) = w$, the time label as $t(e) = t$, and the traversal time as $\lambda(e) = \lambda_e$. For a vertex v , τ_v denotes the set of time steps where v has incoming or outgoing time arcs.

Delays. When a time arc e gets delayed, then its traversal time $\lambda(e)$ is increased by some fixed amount δ . For a given set $D \subseteq E$ of *delayed arcs*, a sequence of time arcs $(v_i, w_i, t_i, \lambda_i)_{i=1}^{\ell}$ is called a *D-delayed temporal walk* if it is a temporal walk in the temporal graph obtained from \mathcal{G} by applying delays to all time arcs in D . (We omit D when it is clear from the context.)



■ **Figure 2** A temporal graph which, for $x = \delta = 1$, forms a yes-instance of DELAYED-ROUTING GAME but a no-instance of DELAYED-ROUTING PATH GAME. In the former, a winning strategy for the traveler is to take the thick blue time arc to a . If it is not delayed, then they can directly go to z . If it is delayed, then there are no remaining delays. Thus, after returning to s , the target vertex z can be reached via b .

As an example consider the temporal walk $(a, b, 1, 1), (b, c, 3, 1)$:



When delaying the first time arc by 1, i.e. having $\delta = 1$ and $D = \{(a, b, 1, 1)\}$, then this is also a delayed temporal walk: Due to the delay, the first time arc arrives in b at time step $2 + \delta = 3$ which is still not later than the departure of the second time arc. However, if we instead set $\delta = 2$ and $D = \{(a, b, 1, 1)\}$, then it is no longer a delayed temporal walk, because the first time arc only reaches b at time 4.

Clearly, from any temporal walk one can obtain a temporal path by eliminating all circular subwalks. Thus, for any delay set D , if there is a D -delayed temporal walk from s to z , then there is also a D -delayed temporal path. This is the reason why we did not define a separate version of DELAY-ROBUST CONNECTION for temporal walks.

Note that this equivalence does not extend to DELAYED-ROUTING GAME as Figure 2 proves.

Static expansion. Sometimes, a problem on a temporal graph \mathcal{G} is transformed to problems on a non-temporal “time-expanded” graph, a *static expansion* of \mathcal{G} . The idea is to model each temporal occurrence of every vertex in the temporal graph as a distinct vertex of the static expansion.

Formally, we say that a digraph $H = (W, A)$ is a *static expansion* of the temporal graph $\mathcal{G} = (V, E)$ if

- (i) $\bigcup_{v \in V} (\{v\} \times \tau_v) \subseteq W \subseteq V \times \mathbb{N}$, and
- (ii) $A = A_1 \cup A_2$ with

$$A_1 = \left\{ (v, t), (v, t') \mid (v, t) \in W \wedge t' = \arg \min_{t'' > t} \{(v, t'') \in W\} \right\},$$

$$A_2 = \{((v, t), (w, t + \lambda)) \mid (v, w, t, \lambda) \in E\}.$$

The arcs in A_1 are often called *waiting arcs* while the arcs in A_2 are in one-to-one correspondence to the time arcs of \mathcal{G} . We call the arcs in A_2 arcs *corresponding* to E . The static expansion with the minimal set of vertices is called the *reduced static expansion*.

The main virtue of static expansions is that they model temporal walks as (non-temporal) paths. More precisely, we have the following lemma.

► **Lemma 1.** *If (v, t) and (w, u) are two vertices of a static expansion H of \mathcal{G} , then there is a path from (v, t) to (w, u) if and only if \mathcal{G} contains a temporal walk from v to w which starts at time t or later and arrives at time u or earlier.*

The proof of Lemma 1 is folklore; we omit it, as well as the proof of the following easy observation.

► **Observation 2.** *If $H = (W, A)$ is a static expansion of $\mathcal{G} = (V, E)$, $E' \subseteq E$ is a set of time arcs and $A' \subseteq A$ the set of arcs corresponding to E' , then $(W, A \setminus A')$ is a static expansion of $(V, E \setminus E')$.*

3 Delay-Robust Connection

In this section, we present an algorithm (Algorithm 1) which solves DELAY-ROBUST CONNECTION in polynomial time. The core idea is to reduce the problem to the computation of a maximum flow problem in a static expansion. There are three steps in this algorithm. First, we construct a new temporal graph \mathcal{G}^* in which the removal of a time arc has the same effect as delaying the respective time arc in the original input graph $\mathcal{G} = (V, E)$. Second, we construct a static expansion H of \mathcal{G}^* . Finally, we compute the maximum flow from the start vertex s to the target vertex z in H . We will show that the value of this flow equals the number of delays required to break temporal connectivity between s and z in \mathcal{G} .

For the first step, define $\mathcal{G}^* = (V, E \cup E^*)$ where $E^* = \{(v, w, t, \lambda + \delta) \mid (v, w, t, \lambda) \in E\}$. Then we can observe the following property of \mathcal{G}^* .

► **Lemma 3.** *Let $\mathcal{G} = (V, E)$ be a temporal graph, $s, z \in V$, $D \subseteq E$, and $\mathcal{G}^* = (V, E \cup E^*)$ as defined above. There is a D -delayed temporal (s, z) -walk in \mathcal{G} if and only if there is a temporal (s, z) -walk in $\mathcal{G}_D^* := (V, (E \setminus D) \cup E^*)$.*

Proof.

(\Rightarrow): Let $W = (e_1, e_2, \dots, e_k)$ be a D -delayed walk from s to z in \mathcal{G} with $e_i = (v_i, w_i, t_i, \lambda_i)$ for $i \in [k]$. This means that $v_1 = s$, $w_k = z$, and for all $\ell \in [k-1]$ it holds that $w_\ell = v_{\ell+1}$ and $t_\ell + \lambda_\ell + [e_\ell \in D] \cdot \delta \leq t_{\ell+1}$. We construct the temporal walk $\hat{W} = (\hat{e}_1, \hat{e}_2, \dots, \hat{e}_k)$ in \mathcal{G}_D^* with $\hat{e}_i = (v_i, w_i, t_i, \lambda_i + [e_i \in D] \cdot \delta)$ for $i \in [k]$. Then $t(\hat{e}_\ell) + \lambda(\hat{e}_\ell) \leq t(\hat{e}_{\ell+1})$ holds for all $\ell \in [k-1]$, thus \hat{W} is a temporal walk from s to z in \mathcal{G}_D^* .

(\Leftarrow): Let $W = (e_1, e_2, \dots, e_k)$ be a temporal (s, z) -walk in \mathcal{G}_D^* with $e_i = (v_i, w_i, t_i, \lambda_i)$. This means that $v_1 = s$, $w_k = z$ and for all $\ell \in [k-1]$, it holds that $w_\ell = v_{\ell+1}$ and $t_\ell + \lambda_\ell \leq t_{\ell+1}$. We construct a delayed temporal walk $\hat{W} = (\hat{e}_1, \hat{e}_2, \dots, \hat{e}_k)$ in \mathcal{G} for the delay set D with

$$\hat{e}_i = \begin{cases} e_i, & \text{if } e_i \in E \setminus D \\ (v_i, w_i, t_i, \lambda_i - \delta), & \text{otherwise} \end{cases}$$

for $i \in [k]$. Note that $\hat{e}_i \in E$: If $e_i \notin E \setminus D$, then $e_i \in E^*$, and thus $\hat{e}_i = (v_i, w_i, t_i, \lambda_i - \delta) \in E$ by construction of E^* . We then have for all $\ell \in [k-1]$ that

$$t(\hat{e}_\ell) + \lambda(\hat{e}_\ell) + [\hat{e}_\ell \in D] \cdot \delta = t(e_\ell) + \lambda(e_\ell) \leq t(e_{\ell+1}) = t(\hat{e}_{\ell+1})$$

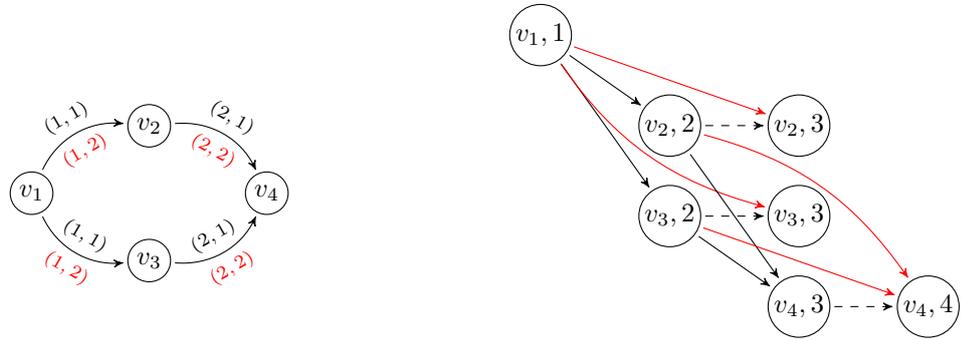
holds, hence \hat{W} is a D -delayed temporal walk in \mathcal{G} . ◀

The algorithm for DELAY-ROBUST CONNECTION is given as pseudo-code in Algorithm 1. See also Figure 3 for an illustration of the graph H . We now show the correctness of the algorithm.

Algorithm 1 DELAY-ROBUST CONNECTION.

Input: $\mathcal{G} = (V, E)$, $s, z \in V$, $x, \delta \in \mathbb{N}$
Output: Is $(\mathcal{G}, s, z, x, \delta)$ a yes-instance of DELAY-ROBUST CONNECTION?

- 1: $H \leftarrow$ reduced static expansion of \mathcal{G}^*
- 2: Define $c : E(H) \rightarrow \{1, \infty\}$ by $c(e) = \begin{cases} 1 & \text{if } e \text{ corresponds to a time arc in } E \\ \infty & \text{otherwise} \end{cases}$
- 3: $T \leftarrow \arg \max_t \{(z, t) \in V(H)\}$
- 4: Compute the value f of a maximum flow from $(s, 0)$ to (z, T) in H with capacities c
- 5: **if** $f \leq x$ **then**
- 6: **return** “NO”
- 7: **else**
- 8: **return** “YES”



(a) The temporal graph \mathcal{G}^* . The black part of the figure shows \mathcal{G} , while the red time arcs are in E^* .

(b) The reduced static expansion of \mathcal{G}^* . The dashed arcs are waiting arcs and the black part forms a static expansion of \mathcal{G} . Note that the capacity function c assigns 1 exactly to the solid black edges.

Figure 3 Example depiction of \mathcal{G}^* and its reduced static expansion ($\delta = 1$).

► **Lemma 4.** *Algorithm 1 solves DELAY-ROBUST CONNECTION.*

Proof. By Lemma 3, the given instance of DELAY-ROBUST CONNECTION is a no-instance if and only if there exists a set $D \subseteq E$ of size $|D| \leq x$ such that \mathcal{G}_D^* contains no temporal (s, z) -walk. By Lemma 1 and Observation 2, this is equivalent of there being a set $D' \subseteq E(H)$ of edges in the static expansion H of \mathcal{G}^* such that

- (i) $|D'| \leq x$ and the edges in D' all correspond to edges in E , and
- (ii) $H - D'$ contains no walk from $(s, 0)$ to (z, T) where T is the largest integer for which H contains the vertex (z, T) .

Note that ii is equivalent to D' forming a cut set that separates $(s, 0)$ and (z, T) . Also, by definition of the capacity function c , i is equivalent to the total capacity $\sum_{e \in D'} c(e)$ being at most x .

Therefore, by the Max-Flow-Min-Cut-Theorem [13, 8], the given instance is a no-instance if and only if the maximum flow from $(s, 0)$ to (z, T) in the graph H with edge capacities c is at most x . ◀

Next, we analyze Algorithm 1’s running time.

► **Lemma 5.** *Algorithm 1 has a running time of $\mathcal{O}(|E|^2)$.*

Proof. In the first step, the algorithm constructs the reduced static expansion H of the temporal graph \mathcal{G}^* . The size of H is in $\mathcal{O}(|E|)$ and the construction can be done in time linear to its size. Constructing c and T can also be done in $\mathcal{O}(|E|)$ time.

Next, we need to compute the value of a maximum flow in H . Actually, it suffices to only test whether that value exceeds x (and we may assume $x \leq |E|$). This test is possible in $\mathcal{O}(|E|^2)$ time, for example by using the classic method of Ford & Fulkerson [13]. ◀

Finally, Lemmas 4 and 5 give us the following theorem.

▶ **Theorem 6.** *DELAY-ROBUST CONNECTION can be solved in $\mathcal{O}(|E|^2)$ time.*

4 Delayed-Routing Games

In this section, we analyze the problems DELAYED-ROUTING GAME and DELAYED-ROUTING PATH GAME, which ask whether a *traveler* can reach their destination when an *adversary* can delay time arcs while the traveler is traversing them. More formally, the traveler and the adversary are players in a given game instance of the two-player game Delayed-Routing Game or Delayed-Routing Path Game, respectively, and we ask whether the traveler has a winning strategy. Starting at a start vertex s at time step 0, the traveler selects an out-going time arc from the current vertex, while the adversary can then delay a selected time arc by δ time units. However, the number of delays of the adversary is limited to x , thus they cannot always apply a delay. The traveler wins when they reach the target vertex z . In the Delayed-Routing Path Game the traveler can visit each vertex at most once, whereas no such restriction applies in the Delayed-Routing Game.

We present a dynamic program to solve DELAYED-ROUTING GAME in $\mathcal{O}(|V| \cdot |E| \cdot x)$ time. We later use a slightly modified version of the algorithm to show that DELAYED-ROUTING PATH GAME is in PSPACE. Furthermore, using a polynomial-time many-one reduction from QBF GAME we prove that DELAYED-ROUTING PATH GAME is PSPACE-hard. Hence, we can conclude that DELAYED-ROUTING PATH GAME is PSPACE-complete.

4.1 A dynamic program for Delayed-Routing Game

A key observation for our dynamic program is that there are only polynomially many game states in DELAYED-ROUTING GAME. Furthermore, the available moves from a node of the game tree¹ and the determination of the winner only depend on the current node/game state and not on its predecessors. Hence, once we have computed whether the traveler has a winning strategy for any given game state, we can save this information in a dynamic programming table.

Let $I = (\mathcal{G} = (V, E), s, z \in V, \delta, x \in \mathbb{N})$ be an instance of DELAYED-ROUTING GAME. Starting at vertex s at time step 0 and with the adversary having a budget of x delays, the goal for the traveler is to reach the target vertex z . On the traveler's turn, they can select a time arc incident to the current vertex that occurs at the current time step or later. The adversary can then decide whether they delay this time arc by δ , thus reducing their number of remaining delays by 1. Once the current vertex is the target vertex z , the game ends with the traveler as the winner. If at any point there are no available time arcs, then the game ends with the adversary as the winner. If the game runs indefinitely, then the adversary also wins the game (since \mathcal{G} is finite, this can only occur if the traveler follows a cycle with traversal time 0).

¹ The *game tree* consists of all possible game states linked by the successor relationship [25].

17:8 Temporal Connectivity: Coping with Foreseen and Unforeseen Delays

At the traveler's turn, the game state can be fully described by the 3-tuple (v, t, y) , where $v \in V$ is the current vertex, $t \in \mathbb{N}$ is the current time step, and $y \in [0, x]$ is the number of remaining delays. We may take t to be from the set $T := \{1, t(e) + \lambda(e), t(e) + \lambda(e) + \delta \mid e \in E\}$. The starting game state is $(s, 1, x)$.

We define a dynamic programming table $F : V \times T \times [-1, x] \rightarrow \{\mathbf{true}, \mathbf{false}\}$ where $F(v, t, y) = \mathbf{true}$ if the traveler has a winning strategy from the game state (v, t, y) . Note that we allow the delay budget to reach -1 for technical reasons, but we will define $F(v, t, -1) = \mathbf{true}$ for all $v \in V, t \in T$. This can be interpreted as allowing the adversary to "cheat" by exceeding their budget of delays, at the cost of immediately losing. Since this option is never beneficial for the adversary, providing it does not change the game in any significant way.

Denote by $E_t(v) := \{(v, w, t', \lambda) \in E \mid t' \geq t\}$ the set of all time arcs that are available at $v \in V$ at time t or later. Then we have the following.

► **Lemma 7.** *For all $v \in V \setminus \{z\}$, $t \in T$, and $y \in [0, x]$ it holds that*

$$F(z, t, y) = \mathbf{true} \tag{1}$$

$$F(v, t, -1) = \mathbf{true} \tag{2}$$

$$F(v, t, y) = \bigvee_{e \in E_t(v)} (F(\text{end}(e), t(e) + \lambda(e), y) \wedge F(\text{end}(e), t(e) + \lambda(e) + \delta, y - 1)) \tag{3}$$

where the empty disjunction evaluates to *false*.

Proof. Equation (1) is trivially correct, since the traveler has reached their destination vertex z . Equation (2) holds by definition as noted above.

It remains to prove (3). By the rules of the game, the traveler may choose any time arc $e \in E_t(v)$ when they are in game state (v, t, y) . If the adversary opts to delay that time arc, then the resulting game state is $F(\text{end}(e), t(e) + \lambda(e) + \delta, y - 1)$. Otherwise, the resulting game state is $F(\text{end}(e), t(e) + \lambda(e), y)$. If, for some $e \in E_t(v)$, the traveler has winning strategies for both of these game states, then they can win from (v, t, y) by picking e and then proceeding from either of the two resulting game states according to their respective winning strategy. Conversely, if all of the time arcs in $E_t(v)$ lead to a game state from which the adversary has a winning strategy, then the traveler clearly cannot win. ◀

In principle, we would like to use Lemma 7 to compute all values of F . However, in the presence of arcs with traversal time zero, Lemma 7 might not suffice to completely determine all values of F . Consider the example instance given in Figure 4. For all $v \in V, t \in T$, and $y \in \{0, 1\}$ we clearly have

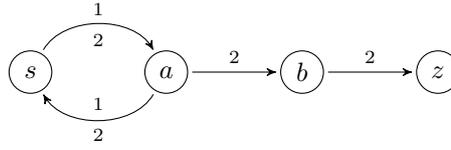
$$F(b, t, y) = [t \leq 2] \quad \text{and} \quad F(a, t, 0) = [t \leq 2]$$

by (3). Note that we can compute

$$F(a, 2, 0) = F(b, 2, 0) \vee F(s, 2, 0) = \mathbf{true} \vee F(s, 2, 0) = \mathbf{true}$$

without needing the value of $F(s, 2, 0)$. However, $F(s, 1, 1)$ and $F(a, 1, 1)$ depend on each other through (3):

$$\begin{aligned} F(s, 1, 1) &= (F(a, 1, 1) \wedge F(a, 2, 0)) \vee (F(a, 2, 1) \wedge F(a, 3, 0)) = F(a, 1, 1) \quad \text{and} \\ F(a, 1, 1) &= (F(s, 1, 1) \wedge F(s, 2, 0)) \vee (F(s, 2, 1) \wedge F(s, 3, 0)) \vee (F(b, 2, 1) \wedge F(b, 3, 0)) \\ &= F(s, 1, 1). \end{aligned}$$



■ **Figure 4** A Delayed-Routing Game instance in which the traveler cycles between s and a forever. All traversal times are 0 (omitted in the figure), and $\delta = x = 1$. Edges with multiple labels occur at multiple times.

If the value of a table entry is not determined through (1)–(3), then we call this value *hung*. As we have seen, it can occur that a subset of all table entries remains hung, even when all other entries have been computed. This is precisely the case when in each clause appearing in the disjunction (3) at least one of the two referenced entries is either **false** or hung itself. In other words, from a hung state, the traveler only has the options to move to a losing state or to another hung state. In particular, the traveler can play on forever but never reach a winning state. Thus, in accordance with our rule that the adversary shall win if the game continues forever, we may set all hung table entries to **false** in this case.

We summarize this in the following lemma.

► **Lemma 8.** *If every value of F has either been computed or depends, through (3), on another still uncomputed value, then the traveler does not have a winning strategy from any of the game states whose values are still uncomputed.*

Next, we determine the time required to fill the dynamic programming table.

► **Lemma 9.** *All entries of F can be computed in $O(|V| \cdot |E| \cdot x)$ time.*

Proof. The number of table entries is $N := |V| \cdot |T| \cdot (x + 2) \in \mathcal{O}(|V| \cdot |E| \cdot x)$. For any $t \in T$, denote by t^+ the smallest element of T strictly larger than t . Begin by observing that (3) can be replaced with the following equivalent formula.

$$F(v, t, y) = F(v, t^+, y) \vee \bigvee_{e \in E_t(v) \setminus E_{t^+}(v)} (F(\text{end}(e), t(e) + \lambda(e), y) \wedge F(\text{end}(e), t(e) + \lambda(e) + \delta, y - 1)) \quad (4)$$

We compute the table entries using Lemma 7 as follows. Start with the entries given by (1). Whenever an entry $F(v, t, y)$ is set to **true**, check whether any of the entries directly depending on $F(v, t, y)$ through (4) can be also computed (i.e., set to **true**, as (4) contains no negations). Note that there are three ways of how another entry $F(v', t', y')$ can directly depend on $F(v, t, y)$:

- (i) $v = v'$, $y = y'$, and $t = t^+$,
- (ii) $y' = y$ and there is a time arc $(v', v, \hat{t}, \lambda)$ with $t' \leq \hat{t} < t^+$ and $\hat{t} + \lambda = t$, or
- (iii) $y' = y + 1$ and there is a time arc $(v', v, \hat{t}, \lambda)$ with $t' \leq \hat{t} < t^+$ and $\hat{t} + \lambda + \delta = t$.

In particular, each time arc causes a direct dependency only between about $2x$ pairs of entries through (ii) and (iii). The number of dependencies through (i) is clearly at most N . Thus, the overall number of checks to be performed is at most $2x \cdot |E| + N$ and each of these checks can be done in constant time.

Afterwards, all remaining entries must be either **false** or hung: Since (4) contains no negations, setting entries to **false** can never cause other entries to become **true**. By Lemma 8, we can thus set all remaining entries to **false**. Hence, the entire table can be filled in $\mathcal{O}(x \cdot |E| + N) \subseteq \mathcal{O}(|V| \cdot |E| \cdot x)$ time. ◀

17:10 Temporal Connectivity: Coping with Foreseen and Unforeseen Delays

To solve DELAYED-ROUTING GAME, we can now evaluate F and check whether $F(s, 1, x) = \text{true}$. Hence, Lemma 9 gives us the following.

► **Theorem 10.** *DELAYED-ROUTING GAME can be solved in $O(|V| \cdot |E| \cdot x)$ time.*

4.2 PSPACE-hardness of Delayed-Routing Path Game

We now present a polynomial-time reduction from the PSPACE-complete QBF GAME to DELAYED-ROUTING PATH GAME. QBF GAME is a game formulation of the problem QBF that asks whether a given quantified boolean formula is true. In the game variant, Player 1 and Player 2 choose truth assignments for existentially and universally quantified variables, respectively. Player 1 wins when the formula is satisfied, otherwise Player 2 wins. If Player 1 has a winning strategy, then it is a yes-instance. QBF and QBF GAME are equivalent and known to be PSPACE-complete [1].

In QBF GAME, we are given a quantified boolean formula $\Phi = Q_1x_1.Q_2x_2.\dots.Q_nx_n.\varphi$ with $Q_i \in \{\exists, \forall\}$ and φ being a boolean formula. The game then consists of n rounds. In the i -th round, if $Q_i = \exists$, then Player 1 selects a truth value for x_i . Else if $Q_i = \forall$, then Player 2 selects a truth value for x_i . If after the n -th round φ is satisfied under the selected truth assignment, then Player 1 wins, otherwise Player 2 is the winner.

QBF GAME

Input: A quantified boolean formula $\Phi = Q_1x_1.Q_2x_2.\dots.Q_nx_n.\varphi$ with $Q_i \in \{\exists, \forall\}$.

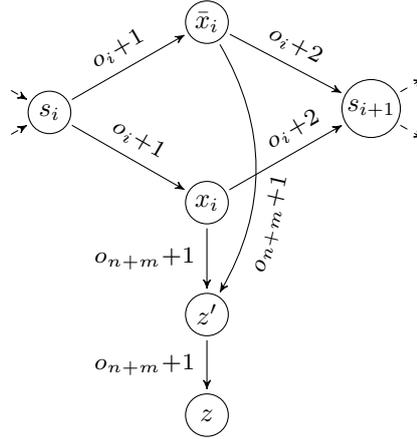
Question: Is there a winning strategy for Player 1?

Given a QBF GAME-instance $\Phi = Q_1x_1.Q_2x_2.\dots.Q_nx_n.\varphi$, we construct an instance $(\mathcal{G} = (V, E), s, z \in V, x, \delta \in \mathbb{N})$ of DELAYED-ROUTING PATH GAME, so that Player 1 has a winning strategy for Φ if and only if the traveler has a winning strategy for the DELAYED-ROUTING PATH GAME-instance. Without loss of generality, we assume that φ is in conjunctive normal form and has three literals per clause.

The main idea for our reduction is the following. The temporal graph \mathcal{G} we create in our reduction consists of n chained selection gadgets, one for each quantified variable, and $m - 1$ validation gadgets where m is the number of clauses of φ . In a selection gadget for a variable quantified by an existential quantifier, the traveler can freely choose one of two paths, corresponding to a truth assignment of this variable. A delay by the adversary has no effect in this gadget. In a selection gadget for a variable quantified by a universal quantifier, a delay of the adversary forces the traveler to take a specific path, corresponding to a truth assignment of this variable. If not taking this enforced path, the traveler gets immediately stuck and loses the game. The gadget is constructed in a way that the adversary needs to use exactly one delay per universal quantified variable. Using no delay lets the traveler immediately win, while using more than one delay is no better for the adversary than a single delay.

In the validation gadgets, the adversary can force the traveler to take one of two paths, one corresponding to selecting the corresponding clause of φ , the other leading to the next validation gadget. In this way, the adversary can select one clause of φ . After traversing either all validation gadgets or selecting a specific clause, the adversary has no remaining delays.

Finally, if there is a literal in the clause which is satisfied under the truth assignment corresponding to the path taken in the selection gadgets, then the traveler can traverse back to a vertex in the selection gadget. From this vertex, the traveler can then reach the target vertex. Otherwise, if all literals in the clause are unsatisfied, then all vertices reachable with a time arc have already been visited when traversing the selection gadgets. Thus the traveler becomes stuck.



■ **Figure 5** Selection gadget for the existentially quantified variable x_i . The traveler can choose freely whether the upper or lower path to s_{i+1} is taken. The adversary has no incentive to apply any delays. Dwelling in x_i or \bar{x}_i to get to z' is no option for the traveler as long as there are delays left.

Now we describe the reduction more formally. All time arcs in the constructed temporal graph \mathcal{G} have a traversal time of 0, thus we write time arcs as 3-tuples $(v, w, t) \in E$ and omit the traversal time in figures. We set the number of delays $x := n' + m - 1$, where $n' \leq n$ is the number of universal quantifiers in Φ . Furthermore, we set $\delta := 1$. The start and target vertices of the game are s_1 and z , respectively, which are added during the construction of the temporal graph. The gadgets use an offset o_i , starting with $o_1 = 0$. The other offsets are computed while constructing the gadgets. Initially, we add the vertices $s_1, s_2, \dots, s_{n+1}, z'$, and z to V , and we add the time arc

$$z' \xrightarrow{o_{n+m}+1} z.$$

The time step $o_{n+m} + 1$ is the largest time step of the constructed temporal graph.

Selection Gadgets

The selection gadget is used to assign a truth value to the quantified variable $Q_i x_i$. The gadget depends on the type of quantifier:

Case 1. $Q_i x_i = \exists x_i$; the variable x_i is existentially quantified.

We add the vertices x_i and \bar{x}_i to V . Furthermore, we add the following time arcs

$$s_i \xrightarrow{o_i+1} x_i \xrightarrow{o_i+2} s_{i+1} \quad \text{and} \quad s_i \xrightarrow{o_i+1} \bar{x}_i \xrightarrow{o_i+2} s_{i+1}.$$

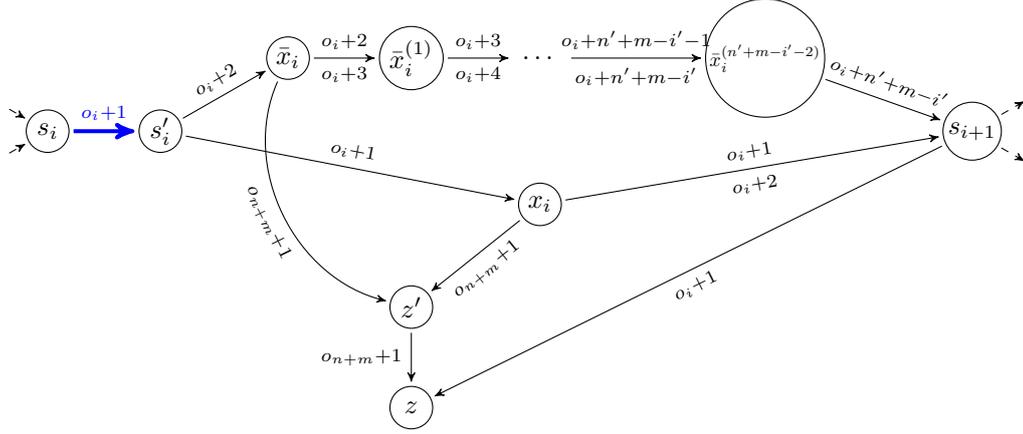
corresponding to assigning x_i to true and false, respectively. The traveler can choose whether to reach s_{i+1} over the vertex x_i or \bar{x}_i . A delay of the adversary will have no effect.

Additionally, we add the time arcs

$$x_i \xrightarrow{o_{m+n}+1} z' \quad \text{and} \quad \bar{x}_i \xrightarrow{o_{m+n}+1} z';$$

however, when traversing this gadget, if the traveler dwells in x_i or \bar{x}_i to take the time arc to z' , then the adversary can use a delay that makes the only outgoing time arc to z at time step $o_{m+n} + 1$ unavailable.

We set the offset $o_{i+1} := o_i + 2$. An example of a selection gadget for existentially quantified variables can be seen in Figure 5.



■ **Figure 6** Selection gadget for the universally quantified variable x_i . Let i' be the number of universally quantified variables before the i -th variable. The traveler enters with $n' + m - i' - 1$ delays left. If the adversary delays the first (thick, blue) time arc, only the upper path to s_{i+1} is available for the traveler. The remaining $n' + m - i' - 2$ delays are not enough to make the traveler stuck, and the adversary has no incentive to use any further delays. If the adversary does not delay the first (thick, blue) time arc, then the traveler is forced to take the lower path to s_{i+1} , since if the adversary applies all $n' + m - i' - 1$ delays on the upper time arcs, then the traveler gets stuck. The adversary will have to apply one delay on the lower time arcs, otherwise the traveler can take the time arc from s_{i+1} to z .

Case 2. $Q_i x_i = \forall x_i$; the variable x_i is universally quantified.

Let i' be the number of universally quantified variables before the i -th variable. We add the vertices s'_i, x_i, \bar{x}_i , and $\bar{x}_i^{(1)}, \bar{x}_i^{(2)}, \dots, \bar{x}_i^{(n'+m-i'-2)}$ to V . Furthermore, we add a time arc $s_i \xrightarrow{o_i+1} s'_i$, and the three time arcs

$$s'_i \xrightarrow{o_i+1} x_i \xrightarrow{o_i+2} s_{i+1},$$

corresponding to setting x_i to true, and

$$s_i \xrightarrow{o_i+2} \bar{x}_i \xrightarrow{o_i+3} \bar{x}_i^{(1)} \xrightarrow{o_i+4} \bar{x}_i^{(2)} \dots \xrightarrow{o_i+n'+m-i'-1} \bar{x}_i^{(n'+m-i'-2)} \xrightarrow{o_i+n'+m-i'} s_{i+1},$$

corresponding to setting x_i to false. Finally, we add a time arc $s_{i+1} \xrightarrow{o_i+1} z$ directly to the end vertex z .

By not delaying the time arc $s_i \xrightarrow{o_i+1} s'_i$, the adversary forces the traveler to take the path through vertex x_i , since the path $s_i \rightarrow \bar{x}_i \rightarrow \bar{x}_i^{(1)} \rightarrow \dots \rightarrow \bar{x}_i^{(n'+m-i'-2)} \rightarrow s_{i+1}$ can be broken by applying all remaining $n' + m - i' - 1$ delays. The adversary is still enforced to apply one delay in $s'_i \rightarrow x_i \rightarrow s_{i+1}$, otherwise the traveler can take $s_{i+1} \xrightarrow{o_i+1} z$ and directly wins. By delaying the time arc $s_i \xrightarrow{o_i+1} s'_i$, the adversary forces the traveler to take the path through vertex \bar{x}_i , since the time arc $s_i \xrightarrow{o_i+1} x_i$ becomes unavailable. However, the remaining $n' + m - i'$ delays are not enough to break the path $s_i \rightarrow \bar{x}_i \rightarrow \bar{x}_i^{(1)} \rightarrow \bar{x}_i^{(2)} \dots \bar{x}_i^{(n'+m-i') \rightarrow s_{i+1}}$.

Additionally, we add the time arcs

$$x_i \xrightarrow{o_{m+n+1}} z' \quad \text{and} \quad \bar{x}_i \xrightarrow{o_{m+n+1}} z',$$

however when traversing this gadget, if the traveler dwells in x_i or \bar{x}_i to take the time arc to z' , then the adversary can use a delay that makes the only outgoing time arc to z at time step $o_{m+n} + 1$ unavailable.

We set the offset $o_{i+1} := o_i + n' + m - i'$. An example of a selection gadget for universally quantified variables can be seen in Figure 6.

Validation Gadgets

The validation gadgets are used to check whether the formula φ is satisfied for the truth assignment chosen in the selection gadgets. By using all $m - 1$ remaining delays, the adversary can force the traveler to visit a vertex corresponding to a specific clause of φ . For the clauses c_1, c_2, \dots, c_{m-1} there is a validation gadget. By placing a single delay, the adversary can force the traveler to take one of two junctions, where one corresponds to selecting the clause, and the other leads to the next validation gadget. (For the $m - 1$ -st validation gadget the other junction corresponds to the m -th clause.) From there, the traveler can reach z only if there is a satisfied literal in the clause.

For each clause $c_i \in \{c_1, c_2, \dots, c_{m-1}\}$, we add the vertices $v_i, v_i^{(l,k)}$ for $k \in [m - i]$, and $v_i^{(r,k)}$ for $k \in [m - i]$. For $i \in [2, m - 1]$, we add the time arc

$$v_{i-1}^{(r, m-(i-1))} \xrightarrow{o_{n+i+1}} v_i,$$

connecting the i -th validation gadget with the $(i - 1)$ -st validation gadget. For $i = 1$, we add the time arc

$$s_{n+1} \xrightarrow{o_{n+1+1}} v_1,$$

connecting the last selection gadget with the first validation gadget.

For the branch corresponding to selecting the i -th clause, we add the time arcs

$$v_i \xrightarrow{o_{n+i+1}} v_i^{(l,1)} \xrightarrow{o_{n+i+2}} v_i^{(l,2)} \xrightarrow{o_{n+i+3}} \dots \xrightarrow{o_{n+i+m-i}} v_i^{(l,m-i)}.$$

Furthermore, for all $v_i^{(l,k)}$ with $k \in [m - i]$, we add a time arc

$$v_i^{(l,k)} \xrightarrow{o_{n+i+k}} z,$$

which enforces the adversary to place delays on all time arcs above, so the traveler cannot directly go to vertex z and win the game. This ensures that all delays are used after reaching $v_i^{(l,m-i)}$, which corresponds to selecting the i -th clause.

For the branch corresponding to not selecting the i -th clause, we add the time arc

$$v_i \xrightarrow{o_{n+i+2}} v_i^{(r,1)},$$

the time arcs

$$v_i^{(r,k)} \xrightarrow[o_{n+i+k+2}]{o_{n+i+k+1}} v_i^{(r,k+1)},$$

for $k \in [m - i - 1]$, and the time arc

$$v_i^{(r,m-i)} \xrightarrow{o_{n+i+m-i+1}} v_i^{(r,m-i+1)}.$$

Delaying all $m - i$ time arcs from v_i to $v_i^{(r,m-i)}$ will break the connection from $v_i^{(r,m-i)} \rightarrow v_i^{(r,m-i+1)}$; however, if there is one delay less, then the adversary does not get any better by using delays. We set the next offset $o_{n+i+1} := o_{n+i} + m - i + 2$.

17:14 Temporal Connectivity: Coping with Foreseen and Unforeseen Delays

Finally, we add time arcs for the literals in the clauses. Let the i -th clause be

$$(l_i^{(1)} \vee l_i^{(2)} \vee l_i^{(3)}).$$

For all $i \in [m-1]$ and all $j \in [3]$, if $l_i^{(j)} = x_k$, then we add the time arc

$$v_i^{(l,m-i)} \xrightarrow{o_{n+m}+1} \bar{x}_k,$$

and if $l_i^{(j)} = \bar{x}_k$, then we add the time arc

$$v_i^{(l,m-i)} \xrightarrow{o_{n+m}+1} x_k,$$

where $k \in [n]$. For the last clause c_m and for all $j \in [3]$, if $l_i^{(j)} = x_k$, we add the time arc

$$v_{m-1}^{(r,2)} \xrightarrow{o_{n+m}+1} \bar{x}_k,$$

and if $l_i^{(j)} = \bar{x}_k$, then we add the time arc

$$v_{m-1}^{(r,2)} \xrightarrow{o_{n+m}+1} x_k,$$

where $k \in [n]$. At this point there are no delays remaining, and for all $k \in [n]$, the time arcs

$$x_k \xrightarrow{o_{n+m}+1} z' \xrightarrow{o_{n+m}+1} z \quad \text{or} \quad \bar{x}_k \xrightarrow{o_{n+m}+1} z' \xrightarrow{o_{n+m}+1} z,$$

which have been added previously in the selection gadgets, can be traversed.

Due to space constraints, we defer a visualization of the validation gadget and the remaining details of the proof which leads to the following theorem to the full version.

► **Theorem 11.** *DELAYED-ROUTING PATH GAME is PSPACE-hard.*

4.3 PSPACE-containment of Delayed-Routing Path Game

Complementing the PSPACE-hardness of DELAYED-ROUTING PATH GAME from the previous section, we now show that DELAYED-ROUTING PATH GAME is contained in PSPACE, which lets us conclude PSPACE-completeness of DELAYED-ROUTING PATH GAME.

We show this by modifying the dynamic program for DELAYED-ROUTING GAME (Section 4.1) to also save the set of vertices that already has been visited for every state. This will cause the dynamic programming table to have exponential size; however, we can evaluate it recursively, that is, recomputing every entry when needed. In this way we only require polynomial space. Formally, we adapt the recursive formula as follows.

Let $I = (G = (V, E), s, z \in V, \delta, x \in \mathbb{N})$ be an instance of DELAYED-ROUTING PATH GAME. A game state can be fully described by the 4-tuple (v, t, y, V') , where $v \in V$ is the current vertex, $t \in \mathbb{N}$ is the current time step, $y \in [0, x]$ is the number of remaining delays, and $V' \subseteq V$ the set of visited vertices. We may take t to be from the set $T := \{1, t(e) + \lambda(e), t(e) + \lambda(e) + \delta \mid e \in E\}$. The starting game state is $(s, 1, x, \emptyset)$.

We define $F : V \times T \times [-1, x] \times 2^V \rightarrow \{\mathbf{true}, \mathbf{false}\}$ to indicate for each game state whether the traveler has a winning strategy from that game state. Denote by $E_t(v) := \{(v, w, t', \lambda) \in E \mid t' \geq t\}$ the set of all time arcs that are available at $v \in V$ at time t or later. Then, for all $v \in V \setminus \{z\}$, $t \in T$, $y \in [0, x]$, and $V' \subseteq V \setminus \{z\}$ the following holds:

$$F(z, t, y, V') = \mathbf{true}, \quad (5)$$

$$F(v, t, -1, V') = \mathbf{true}, \quad (6)$$

$$F(v, t, y, V') = \bigvee_{e \in E_t(v)} (\text{end}(e) \notin V' \wedge F(\text{end}(e), t(e) + \lambda(e), y, V' \cup \{v\}) \wedge F(\text{end}(e), t(e) + \lambda(e) + \delta, y - 1, V' \cup \{v\})), \quad (7)$$

where the empty disjunction evaluates to **false**.

Using (5)–(7), we get the following result by evaluating it in a depth-first-search fashion from the starting configuration.

► **Proposition 12.** *DELAYED-ROUTING PATH GAME is contained in PSPACE.*

Proof. The correctness of our approach can be shown in a way analogous to Lemma 7. Note that since the set V' of visited vertices is growing with each move of the traveler, the game cannot run infinitely. Thus, all entries can be computed by means of (5)–(7).

Instead of storing all (exponentially many) entries, we evaluate F in a depth-first-search fashion from the starting configuration $(s, 1, x, \emptyset)$. This only requires us to keep the current branch of the search tree in memory. Since each game state requires polynomial space and there are at most $O(|V|)$ moves in a game of DELAYED-ROUTING PATH GAME (every vertex can be visited at most once), we only require polynomial space. ◀

From Theorem 11 and Proposition 12 we can now conclude that DELAYED-ROUTING PATH GAME is PSPACE-complete.

► **Corollary 13.** *DELAYED-ROUTING PATH GAME is PSPACE-complete.*

5 Conclusion and Outlook

On the spectrum of delay-related routing problems, we have studied two extreme (but natural) cases in terms of when information about the delays is made available. Interestingly, both are polynomial-time solvable, whereas a “middle ground” case studied in companion work turned out NP-hard.

It might also seem surprising that DELAYED-ROUTING GAME is efficiently solvable while DELAYED-ROUTING PATH GAME is PSPACE-complete. However, this situation is not unprecedented. For example, deciding whether a temporal path under waiting time constraints exists (Δ -RESTLESS TEMPORAL PATH) is NP-complete [6], while finding temporal walks under waiting time constraints can be done in polynomial time [2]. Similarly, counting foremost temporal paths is #P-hard [23], while counting of foremost temporal walks can be done in polynomial time [24].

We remark that instead of delaying edges by increasing their traversal time, it is also sensible to instead delay their time label. It can be shown that our results on DELAY-ROBUST CONNECTION transfer also to this modified version. For DELAYED-ROUTING GAME the situation is more complicated, we leave this open for future work.

Even more different notions of delays could also be explored. While in our definitions up to $x \in \mathbb{N}$ time arcs can be delayed by a fixed integer δ each, one could also define an overall “budget” Δ which can be distributed among all time arcs. Thus, a time arc could be delayed by more than δ or more than x time arcs could be delayed by less than δ each.

References

- 1 Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.
- 2 Matthias Bentert, Anne-Sophie Himmel, André Nichterlein, and Rolf Niedermeier. Efficient computation of optimal temporal walks under waiting-time constraints. *Applied Network Science*, 5(1):73, 2020. doi:10.1007/s41109-020-00311-0.
- 3 Binh-Minh Bui-Xuan, Afonso Ferreira, and Aubin Jarry. Computing shortest, fastest, and foremost journeys in dynamic networks. *International Journal of Foundations of Computer Science*, 14(02):267–285, 2003. doi:10.1142/S0129054103001728.
- 4 Sebastian Buß, Hendrik Molter, Rolf Niedermeier, and Maciej Rymar. Algorithmic aspects of temporal betweenness. In *Proceedings of the 26th SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pages 2084–2092, 2020. doi:10.1145/3394486.3403259.
- 5 Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012. doi:10.1080/17445760.2012.668546.
- 6 Arnaud Casteigts, Anne-Sophie Himmel, Hendrik Molter, and Philipp Zschoche. Finding temporal paths under waiting time constraints. *Algorithmica*, 83(9):2754–2802, 2021. doi:10.1007/s00453-021-00831-w.
- 7 Argyrios Deligkas and Igor Potapov. Optimizing reachability sets in temporal graphs by delaying. In *Proceedings of the 34th Conference on Artificial Intelligence (AAAI)*, pages 9810–9817, 2020. doi:10.1609/aaai.v34i06.6533.
- 8 Peter Elias, Amiel Feinstein, and Claude E. Shannon. A note on the maximum flow through a network. *IRE Transactions on Information Theory*, 2(4):117–119, 1956. doi:10.1109/TIT.1956.1056816.
- 9 Jessica Enright, Kitty Meeks, George B. Mertzios, and Viktor Zamaraev. Deleting edges to restrict the size of an epidemic in temporal networks. *Journal of Computer and System Sciences*, 119:60–77, 2021. doi:10.1016/j.jcss.2021.01.007.
- 10 Jessica Enright, Kitty Meeks, and Fiona Skerman. Assigning times to minimise reachability in temporal graphs. *Journal of Computer and System Sciences*, 115:169–186, 2021. doi:10.1016/j.jcss.2020.08.001.
- 11 Thomas Erlebach, Michael Hoffmann, and Frank Kammer. On temporal graph exploration. *Journal of Computer and System Sciences*, 119:1–18, 2021. doi:10.1016/j.jcss.2021.01.005.
- 12 Till Fluschnik, Hendrik Molter, Rolf Niedermeier, Malte Renken, and Philipp Zschoche. Temporal graph classes: A view through temporal separators. *Theoretical Computer Science*, 806:197–218, 2020. doi:10.1016/j.tcs.2019.03.031.
- 13 L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956. doi:10.4153/CJM-1956-045-5.
- 14 Eugen Füchle, Hendrik Molter, Rolf Niedermeier, and Malte Renken. Delay-robust routes in temporal graphs. In *39th International Symposium on Theoretical Aspects of Computer Science (STACS 2022)*, volume 219 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:15, 2022. doi:10.4230/LIPIcs.STACS.2022.30.
- 15 Eugen Füchle, Hendrik Molter, Rolf Niedermeier, and Malte Renken. Temporal connectivity: Coping with foreseen and unforeseen delays, 2022. arXiv:2201.05011.
- 16 Petter Holme. Modern temporal network theory: a colloquium. *The European Physical Journal B*, 88(9):234, 2015. doi:10.1140/epjb/e2015-60657-4.
- 17 Petter Holme and Jari Saramäki, editors. *Temporal Network Theory*. Springer, 2019. doi:10.1007/978-3-030-23495-9.
- 18 David Kempe, Jon Kleinberg, and Amit Kumar. Connectivity and inference problems for temporal networks. *Journal of Computer and System Sciences*, 64(4):820–842, 2002. doi:10.1006/jcss.2002.1829.
- 19 Nina Klobas, George B. Mertzios, Hendrik Molter, Rolf Niedermeier, and Philipp Zschoche. Interference-free walks in time: Temporally disjoint paths. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 4090–4096, 2021. doi:10.24963/ijcai.2021/563.

- 20 Matthieu Latapy, Tiphaine Viard, and Clémence Magnien. Stream graphs and link streams for the modeling of interactions over time. *Social Network Analysis and Mining*, 8(1):61, 2018. doi:10.1007/s13278-018-0537-7.
- 21 George B Mertzios, Othon Michail, and Paul G Spirakis. Temporal network optimization subject to connectivity constraints. *Algorithmica*, 81(4):1416–1449, 2019. doi:10.1007/s00453-018-0478-6.
- 22 Hendrik Molter, Malte Renken, and Philipp Zschoche. Temporal reachability minimization: Delaying vs. deleting. In *Proceedings of the 46th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 76:1–76:15, 2021. doi:10.4230/LIPIcs.MFCS.2021.76.
- 23 Amir Afrasiabi Rad, Paola Flocchini, and Joanne Gaudet. Computation and analysis of temporal betweenness in a knowledge mobilization network. *Computational Social Networks*, 4(1):1–22, 2017. doi:10.1186/s40649-017-0041-7.
- 24 Maciej Rymar, Hendrik Molter, André Nichterlein, and Rolf Niedermeier. Towards classifying the polynomial-time solvability of temporal betweenness centrality. In *Proceedings of the 47th International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, pages 219–231, 2021. doi:10.1007/978-3-030-86838-3_17.
- 25 Aaron N. Siegel. *Combinatorial game theory*. American Mathematical Society, 2013.
- 26 Huanhuan Wu, James Cheng, Yiping Ke, Silu Huang, Yuzhen Huang, and Hejun Wu. Efficient algorithms for temporal path computation. *IEEE Transactions on Knowledge and Data Engineering*, 28(11):2927–2942, 2016. doi:10.1109/TKDE.2016.2594065.
- 27 Philipp Zschoche, Till Fluschnik, Hendrik Molter, and Rolf Niedermeier. The complexity of finding small separators in temporal graphs. *Journal of Computer and System Sciences*, 107:72–92, 2020. doi:10.1016/j.jcss.2019.07.006.

Fully Dynamic Four-Vertex Subgraph Counting

Kathrin Hanauer  

Faculty of Computer Science, University of Vienna, Austria

Monika Henzinger  

Faculty of Computer Science, University of Vienna, Austria

Qi Cheng Hua

Faculty of Computer Science, University of Vienna, Austria

Abstract

This paper presents a comprehensive study of algorithms for maintaining the number of all connected four-vertex subgraphs in a dynamic graph. Specifically, our algorithms maintain the number of paths of length three in deterministic amortized $\mathcal{O}(m^{\frac{1}{2}})$ update time, and any other connected four-vertex subgraph which is not a clique in deterministic amortized update time $\mathcal{O}(m^{\frac{2}{3}})$. Queries can be answered in constant time. We also study the query times for subgraphs containing an arbitrary edge that is supplied only with the query as well as the case where only subgraphs containing a vertex s that is fixed beforehand are considered. For length-3 paths, paws, 4-cycles, and diamonds our bounds match or are not far from (conditional) lower bounds: Based on the OMv conjecture we show that any dynamic algorithm that detects the existence of paws, diamonds, or 4-cycles or that counts length-3 paths takes update time $\Omega(m^{1/2-\delta})$.

Additionally, for 4-cliques and all connected induced subgraphs, we show a lower bound of $\Omega(m^{1-\delta})$ for any small constant $\delta > 0$ for the amortized update time, assuming the static combinatorial 4-clique conjecture holds. This shows that the $\mathcal{O}(m)$ algorithm by Eppstein et al. [9] for these subgraphs cannot be improved by a polynomial factor.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Theory of computation → Dynamic graph algorithms

Keywords and phrases Dynamic Graph Algorithms, Subgraph Counting, Motif Search

Digital Object Identifier 10.4230/LIPIcs.SAND.2022.18

Related Version *Full Version*: <https://arxiv.org/abs/2106.15524> [11]

Funding This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (Grant agreement No. 101019564, “The Design of Modern Fully Dynamic Data Structures (MoDynStruct)”), as well as from the Austrian Science Fund (FWF) and netIDEE SCIENCE project P 33775-N.



Acknowledgements The authors want to thank Leonhard Paul Sidl for careful proofreading.

1 Introduction

Detecting or counting subgraphs is an important question in social network analysis, where dense subgraphs usually represent communities, as well as in telecommunication network surveillance, and computational biology. This can also be seen in a recent study by Sahu et al. [21]: finding and counting fixed subgraphs was the fourth most popular graph computation in practice, only superseded by finding connected components, computing shortest paths, and answering queries about the degree of neighbors. Furthermore the same study showed that the dynamic setting is important in practice as 65% of the graphs were dynamic. Thus, the goal of this paper is to advance the study of subgraph counting problems in dynamic graphs.



© Kathrin Hanauer, Monika Henzinger, and Qi Cheng Hua;
licensed under Creative Commons License CC-BY 4.0

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).

Editors: James Aspnes and Othon Michail; Article No. 18; pp. 18:1–18:17

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Algorithmic problems in dynamic graphs are usually modeled by the following data structure question. Given a potentially non-empty initial graph and a fixed subgraph pattern \mathcal{P} (such as a k -clique) maintain a data structure that allows the following updates to the current graph G :

- $\text{Insert}(u, v)$: Insert the edge $\{u, v\}$ into G .
- $\text{Delete}(u, v)$: Delete the edge $\{u, v\}$ from G .
- $\text{Query}()$: Return the number of subgraphs of pattern \mathcal{P} in G .

Given a subgraph pattern \mathcal{P} that is not a clique, there are two variants of this problem: One variant, called *induced subgraph counting*, counts a subgraph if it is exactly equivalent to \mathcal{P} and does *not* contain any additional edges. In the *non-induced* version, a subgraph is counted if it contains \mathcal{P} and potentially additional edges (but *not* additional vertices). Eppstein and Spiro [10] studied subgraph counting for all possible connected three-vertex patterns in both the induced and the non-induced variant and gave a dynamic algorithm with amortized update time $\mathcal{O}(h)$, where h is the *h -index* of G , i.e., the maximum number such that the graph contains h vertices of degree at least h . Note that h is $\mathcal{O}(\sqrt{m})$, where m always is the current number of edges in the graph.

There are six connected graphs on four vertices, which we refer to as *length-3 path* P_3 , *claw* C_3 , *paw* P_3 , *4-cycle* C_4 , *diamond* D_4 , and *4-clique* K_4 . Eppstein et al. [9] extended the method of [10] to maintain counts of any (induced and non-induced) connected four-vertex subgraph in amortized time $\mathcal{O}(h^2) = \mathcal{O}(m)$ and $\mathcal{O}(mh^2)$ space. This paper contains a comprehensive study of the complexity of dynamically counting all possible connected four-vertex subgraphs. We present new improved dynamic algorithms and give the first conditional lower bounds.

Upper bounds. We show how to maintain the number of any connected four-vertex non-induced subgraph that is not a clique (such as a paw, a 4-cycle, or a diamond) in update time $\mathcal{O}(m^{2/3})$ and at most $\mathcal{O}(nm)$ space. For graphs with an h -index larger than $\mathcal{O}(m^{1/3})$, our algorithms are hence faster than the $\mathcal{O}(h^2)$ algorithm by Eppstein et al. [9]. Besides, our data structure can also be used to count all s -triangles, i.e., triangles that contain a fixed vertex s , in $\mathcal{O}(m^{1/2})$ update time, constant query time, and $\mathcal{O}(n)$ space, and likewise for s -length-3 paths. The update time is in $\mathcal{O}(m^{2/3})$ for s -4-cycles, s -paws, and s -diamonds, with $\mathcal{O}(n^2)$ space, and $\mathcal{O}(m)$ for s -4-cliques with constant space. For $\varepsilon \in [0, 1]$, our data structure supports queries on the number of triangles containing an arbitrary vertex or edge in $\mathcal{O}(\min(m^{2\varepsilon}, n^2))$ or $\mathcal{O}(\min(m^{1-\varepsilon}, n))$ worst-case time, respectively, with an update time of $\mathcal{O}(m^{\max(\varepsilon, 1-\varepsilon)})$ or $\mathcal{O}(m^\varepsilon)$, respectively, and $\mathcal{O}(n^2)$ space. We also show how to maintain length-3 paths in time $\mathcal{O}(m^{1/2})$, but this result was already stated in [9]. See Table 1 for an overview. All our algorithms are deterministic and the running time bounds are amortized unless stated otherwise.

Lower bounds. We also give the first conditional lower bounds for counting various four-vertex subgraphs based on two popular hypotheses: the Online Boolean Matrix-Vector Multiplication (OMv) conjecture [12] and the Combinatorial k -Clique hypothesis. In the OMv conjecture we are given a Boolean $n \times n$ matrix M that can be preprocessed. Then, an online sequence of vectors v_1, \dots, v_n is presented and the goal is to compute each Boolean product Mv_i (using conjunctions and disjunctions) before seeing the next vector v_{i+1} .

► **Conjecture 1 (OMv).** *For any constant $\delta > 0$, there is no $\mathcal{O}(n^{3-\delta})$ -time algorithm that solves OMv with error probability at most $1/3$ in the word-RAM model with $\mathcal{O}(\log n)$ bit words.*

■ **Table 1** Upper and conditional lower bounds on the time per update and query for counting different subgraphs, where $\delta > 0$ is an arbitrarily small constant and $h \in \mathcal{O}(\sqrt{m})$. Update times are amortized, query times are worst-case. Results in blue are new or improved.

* Read: For polynomial preprocessing time and $\mathcal{O}(\cdot)$ query time, the update time is $\Omega(\cdot)$.

† The previous space complexity for 3-cycles and length-3 paths was $\mathcal{O}(mh)$ with amortized update time $\mathcal{O}(h)$ [10] and $\mathcal{O}(m)$ with amortized update time $\mathcal{O}(m^{\frac{1}{2}})$ for 3-cycles [14]; for (other) 4-vertex subgraphs, it was $\mathcal{O}(mh^2)$ with amortized update time $\mathcal{O}(h^2)$ [9].

^a Thm 4, ^b Thm 5, ^c Cor. 19, ^d Thm 20, ^e Thm 21, ^f Cor. 25, ^g Thm 26, ^{\alpha} [10], ^{\beta} [9], ^{\zeta} [7].

Subgraph	Lower Bounds*		Update Time		Query Time		Space†
	Update	Query	ours	previous	all	$e \in E$	
<i>Non-induced subgraphs and s-subgraphs</i>							
connected, $n = 4$			$\mathcal{O}(m)^a$		$\mathcal{O}(1)$	$\mathcal{O}(1)^a$	$\mathcal{O}(1)/\mathcal{O}(m)^a$
claw \wedge	$\Omega(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)^{\alpha\beta}$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
length-3 path \sphericalcap	$\Omega(m^{\frac{1}{2}-\delta})^g$	$\mathcal{O}(m^{1-\delta})^g$	$\mathcal{O}(m^{\frac{1}{2}})^b$	$\mathcal{O}(h)^{\alpha\beta}$	$\mathcal{O}(1)$	$\mathcal{O}(m^{\frac{1}{2}})^b$	$\mathcal{O}(\min(n^2, m^{1.5}))^b$
paw \sphericalcap	$\Omega(m^{\frac{1}{2}-\delta})^g$	$\mathcal{O}(m^{1-\delta})^g$	$\mathcal{O}(m^{\frac{2}{3}})^b$	$\mathcal{O}(h^2)^\beta$	$\mathcal{O}(1)$	$\mathcal{O}(m^{\frac{2}{3}})^b$	$\mathcal{O}(n^2)^b$
3-cycle \triangle	$\Omega(m^{\frac{1}{2}-\delta})^g$	$\mathcal{O}(m^{1-\delta})^g$	$\mathcal{O}(m^{\frac{1}{2}})^c$	$\mathcal{O}(h)^\alpha$	$\mathcal{O}(1)$	$\mathcal{O}(m^{\frac{1}{2}})^c$	$\mathcal{O}(\min(n^2, m^{1.5}))^c$
4-cycle \diamond	$\Omega(m^{\frac{1}{2}-\delta})^g$	$\mathcal{O}(m^{1-\delta})^g$	$\mathcal{O}(m^{\frac{2}{3}})^b$	$\mathcal{O}(h^2)^\beta$	$\mathcal{O}(1)$	$\mathcal{O}(m^{\frac{2}{3}})^b$	$\mathcal{O}(n^2)^b$
k -cycle, $k \geq 5$	$\Omega(m^{\frac{1}{2}-\delta})^g$	$\mathcal{O}(m^{1-\delta})^g$					
diamond \diamond	$\Omega(m^{\frac{1}{2}-\delta})^g$	$\mathcal{O}(m^{1-\delta})^g$	$\mathcal{O}(m^{\frac{2}{3}})^b$	$\mathcal{O}(h^2)^\beta$	$\mathcal{O}(1)$	$\mathcal{O}(m^{\frac{2}{3}})^b$	$\mathcal{O}(\min(nm, m^{\frac{5}{3}}))^b$
4-clique \boxtimes	$\Omega(m^{1-\delta})^e$	$\mathcal{O}(m^{2-\delta})^e$	$\mathcal{O}(m)^\zeta$	$\mathcal{O}(h^2)^\beta$	$\mathcal{O}(1)$	$\mathcal{O}(m)^\zeta$	$\mathcal{O}(1)$
<i>s-subgraphs</i>							
s-claw \wedge	$\Omega(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)^d$		$\mathcal{O}(1)^d$		$\mathcal{O}(1)^d$
s-length-3-path \sphericalcap	$\Omega(m^{\frac{1}{2}-\delta})^g$	$\mathcal{O}(m^{1-\delta})^g$	$\mathcal{O}(m^{\frac{1}{2}})^d$		$\mathcal{O}(1)^d$		$\mathcal{O}(n)^d$
s-paw \sphericalcap	$\Omega(m^{\frac{1}{2}-\delta})^g$	$\mathcal{O}(m^{1-\delta})^g$	$\mathcal{O}(m^{\frac{2}{3}})^d$		$\mathcal{O}(1)^d$		$\mathcal{O}(n^2)^d$
s-3-cycle \triangle	$\Omega(m^{\frac{1}{2}-\delta})^g$	$\mathcal{O}(m^{1-\delta})^g$	$\mathcal{O}(m^{\frac{1}{2}})^d$		$\mathcal{O}(1)^d$		$\mathcal{O}(n)^d$
s-4-cycle \diamond			$\mathcal{O}(m^{\frac{2}{3}})^d$		$\mathcal{O}(1)^d$		$\mathcal{O}(n^2)^d$
s- k -cycle, $k \geq 5$, odd	$\Omega(m^{\frac{1}{2}-\delta})^g$	$\mathcal{O}(m^{1-\delta})^g$					
s-diamond \diamond	$\Omega(m^{\frac{1}{2}-\delta})^g$	$\mathcal{O}(m^{1-\delta})^g$	$\mathcal{O}(m^{\frac{2}{3}})^d$		$\mathcal{O}(1)^d$		$\mathcal{O}(n^2)^d$
s-4-clique \boxtimes	$\Omega(m^{\frac{1}{2}-\delta})^g$	$\mathcal{O}(m^{1-\delta})^g$	$\mathcal{O}(m)^d$		$\mathcal{O}(1)^d$		$\mathcal{O}(1)^d$
<i>Induced subgraphs</i>							
connected, $n = 4$	$\Omega(m^{1-\delta})^e$	$\mathcal{O}(m^{2-\delta})^e$	$\mathcal{O}(m)^f$	$\mathcal{O}(h^2)^\beta$	$\mathcal{O}(1)$		$\mathcal{O}(1)^f$

Based on the OMv conjecture we show that *detecting* (with probability at least $2/3$ in the word-RAM model with $\mathcal{O}(\log n)$ bit words) the existence of (non-induced) paws, diamonds, 4-cliques, or k -cycles for any $k \geq 3$ in a graph with edge insertions and deletions takes amortized update time $\Omega(m^{1/2-\delta})$ or query time $\Omega(m^{1-\delta})$ if only polynomial preprocessing time is allowed. This lower bound applies also to the worst-case update time of any insertions-only or deletions-only algorithm. Note that this lower bound does not only apply to *counting* the number of such subgraphs but already to *detecting* whether such a subgraph exists. Let s be a fixed vertex in the graph. The same lower bounds apply to algorithms that detect whether a diamond, 4-clique, or k -cycle with odd k containing s exists. Finally, we also show a lower bound for *counting* the number of length-3 paths and length-3 s -paths. We remark that the conditional lower bounds for (s -)3-cycles were already known before [12].

We also use the Combinatorial k -Clique hypothesis which is defined as follows and has become popular in recent years (e.g. [20, 1, 5, 4]).

► **Conjecture 2** (Combinatorial k -Clique). *For any constant $\delta > 0$, for an n -vertex graph there is no $\mathcal{O}(n^{k-\delta})$ time combinatorial algorithm for k -clique detection with error probability at most $1/3$ in the word-RAM model with $\mathcal{O}(\log n)$ bit words.*

Let $\delta > 0$ be a small constant. Based on the 4-clique conjecture we show that (with probability at least $2/3$ in the word-RAM model with $O(\log n)$ bit words) there does not exist a combinatorial algorithm that counts *any connected induced four-vertex subgraph* in a dynamic graph with amortized update time $O(n^{4-2\delta}/m)$, which is $O(m^{1-\delta})$, and query preprocessing time $O(n^{4-2\delta})$. This bound applies also to any insertions-only algorithm. The bound can be extended to any k -clique with $k > 4$ showing that the amortized update time is $\Omega(n^{k-2\delta}/m)$ with $\Omega(n^{k-2\delta})$ preprocessing and query time.

Technical contribution. For the upper bounds we extend and improve upon Eppstein et al. [9] both with respect to running time and space. The high-level idea is as follows: We partition the vertices into (few) *high-degree* and (many) *low-degree* vertices and then maintain for each vertex, vertex pair, or vertex triple certain information in a data structure such as the number of certain paths up to length 3 that contain low-degree vertices. When an edge $\{u, v\}$ is updated, four-vertex subgraphs that contain u, v , and two other low-degree vertices can be quickly counted using the information in the data structure. On the other side, subgraphs that contain a high-degree vertex in addition to u and v can often be counted “from scratch” after each update as there are few high-degree vertices. The more challenging case is the situation where relationships involving two or more high-degree vertices in the subgraph need to be checked or maintained. How to deal with this depends on the subgraph to count. For diamonds, e.g., this requires to keep certain information about triples of vertices.

For the conditional lower bounds based on the combinatorial k -clique conjecture we first directly deduce the lower bound for incremental 4-clique counting. Then we use the fact that (a) we have a lower bound for 4-cliques, (b) we developed algorithms with $\mathcal{O}(m^{2/3})$ update time for all non-induced subgraphs, and (c) there exist “counting formulas” that allow to compute the number of any induced subgraph based on the number of 4-cliques and the number of non-induced subgraphs. Thus, if the number of an induced subgraph pattern could be computed in $\mathcal{O}(m^{1-\delta})$ time per update for some small $\delta > 0$ then we could use the corresponding counting formula and our algorithms for non-induced subgraphs to dynamically maintain the number of 4-cliques, contradicting our dynamic lower bound for 4-cliques.

For the conditional lower bounds based on the OMv conjecture we construct for each subgraph pattern \mathcal{P} based on an 1-uMv instance (which is a variant of OMv) a suitable graph based on \mathcal{P} with $\mathcal{O}(n)$ vertices and $\mathcal{O}(n^2)$ edges such that detecting the existence of the (non-induced) version of \mathcal{P} in the graph equals finding the answer for the 1-uMv instance. Then the lower bound follows as in [12]. The challenge is to construct such a graph. We show how to do this for detecting non-induced (s)-paws, (s)-diamonds, (s)-4-cliques, and (s)- k -cycles for $k \geq 3$ and for counting non-induced length-3 (s)-paths.

Our paper gives in Section 2 the preliminaries, in Section 3 our new algorithms, and in Section 4 our lower bounds. Some proofs had to be omitted due to space and are given in the full version [11].

2 Preliminaries

Basic Definitions

We consider an undirected dynamic graph $G = (V, E)$ and use n to denote the number of vertices and m for the current number of edges. Two vertices $u \neq v$ are *adjacent* if there is an edge $e = \{u, v\} \in E$. In this case, u and v are *incident* to e . The *neighborhood* $N(v)$ of

a vertex v is defined as $\{u \mid \{u, v\} \in E\}$ and v 's *degree* is $\deg(v) = |N(v)|$. As a shorthand notation to exclude just one vertex, we use $N_{\bar{w}}(v) = N(v) \setminus \{w\}$ and $\deg_{\bar{w}}(v) = |N_{\bar{w}}(v)|$, i.e., $\deg_{\bar{w}}(v) = \deg(v) - 1$ if $w \in N(v)$ and $\deg_{\bar{w}}(v) = \deg(v)$ otherwise. A k -*path* (also length- k path) is a sequence of distinct edges $\langle \{v_0, v_1\}, \{v_1, v_2\}, \dots, \{v_{k-1}, v_k\} \rangle$ of length k , where $v_i \neq v_j$ for all $0 \leq i, j \leq k$. A k -*cycle* is a k -path where as an only exception the first vertex equals the last, i.e., $v_0 = v_k$. A 3-cycle is also called *triangle*. A *claw* is a graph consisting of a vertex x , called the *central* vertex, and three edges incident to it. A *paw* is a graph consisting of a triangle together with an additional edge attached to one of the vertices of the triangle. This vertex is called the *central* vertex of the paw and the additional edge the *arm*. A *diamond* is a 4-cycle with a *chord*, i.e., an additional edge connecting one of the two pairs of non-adjacent vertices, which creates two triangles sharing the chordal edge. A k -*clique* is the complete graph K_k on k vertices.

A graph $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. A subgraph $G' = (V', E')$ is said to be *induced* if $E' = \{\{u, v\} \in E \mid u, v \in V'\}$. The term *non-induced subgraph* or just *subgraph* without an adjective refers to all subgraphs, induced and not. For a static graph \mathcal{P} , also called *pattern*, we denote by $\mathbf{c}(G, \mathcal{P})$ the number of (non-induced) subgraphs of G that are isomorphic to \mathcal{P} , and by $\mathbf{c}_I(G, \mathcal{P})$ the number of induced subgraphs that are isomorphic to \mathcal{P} , in each case divided by the number of automorphisms of \mathcal{P} . For a vertex $s \in V$, we further denote by $\mathbf{c}(G, \mathcal{P}, s)$ the number of non-induced subgraphs of G that are isomorphic to \mathcal{P} and contain s . We denote by \mathbb{P}_k the set of connected subgraphs on k vertices. There are six connected graphs in \mathbb{P}_4 , which we refer to as *length-3 path* \curvearrowright , *claw* \wedge , *paw* \curvearrowleft , *4-cycle* \diamond , *diamond* \diamondsuit , and *4-clique* \boxtimes .

Given a pattern \mathcal{P} , we study the problem of *maintaining* the number of occurrences of \mathcal{P} as a subgraph or induced subgraph of G , called the *non-induced subgraph count* $\mathbf{c}(G, \mathcal{P})$ or the *induced subgraph count* $\mathbf{c}_I(G, \mathcal{P})$, respectively, of \mathcal{P} in G , and the analogous problem of *maintaining* the count of non-induced subgraphs containing a specific predefined vertex $v \in V$ or edge $e \in E$, $\mathbf{c}(G, \mathcal{P}, v)$ or $\mathbf{c}(G, \mathcal{P}, e)$, respectively. Unless stated otherwise, *maintaining* a count implies that we can retrieve it by a query in constant time. We also study the closely related problem of *querying* the number of subgraphs containing a specific vertex or edge that is given only with the query.

Further Related Work

Detecting (or counting) subgraphs, also known as the *subgraph isomorphism* problem, generalizes the *clique* or *Hamiltonian cycle* problem and is hence \mathcal{NP} -hard. Nevertheless, it can be solved efficiently if the subgraphs to detect or count are restricted.

Static algorithms. Algorithms counting numbers of subgraphs and solving related problems have been studied extensively for static graphs. Alon, Yuster and Zwick [2] developed an algorithm to count the number of triangles and other circles (up to seven vertices) in a graph in time $\mathcal{O}(n^\omega)$, where n denotes the number of vertices and $\omega < 2.373$ [19] is the fast matrix multiplication exponent, i.e., the smallest value such that two $n \times n$ matrices can be multiplied in $\mathcal{O}(n^\omega)$ time. Kloks, Kratsch and Müller [16] showed how to compute the number of 4-cliques in time $\mathcal{O}(m^{(\omega+1)/2})$ (m denotes the number of edges) and the number of any other subgraph of size 4 in time $\mathcal{O}(n^\omega + m^{(\omega+1)/2})$. There is also a large body of work on parallel algorithms for counting subgraphs (see e.g. [18]) and to develop approximate algorithms in the streaming setting (see e.g. [6, 3]).

Dynamic algorithms. A more recent development is counting subgraph numbers for dynamic graphs. Kara et al. [14] provided an algorithm for counting triangles in amortized time $\mathcal{O}(\sqrt{m})$ per update and $\mathcal{O}(m)$ space, which can also enumerate them with constant time delay. Dhulipala et al. [7] extended it to a batch-dynamic parallel algorithm with $\mathcal{O}(\Delta\sqrt{\Delta+m})$ amortized work and $\mathcal{O}(\text{polylog}(\Delta+m))$ depth *w.h.p.* for a batch of Δ updates. Based on a static algorithm to enumerate cliques, they show how to obtain a dynamic algorithm for maintaining the number of k -cliques for a fixed $k > 3$ with expected $\mathcal{O}(\Delta(m+\Delta)\alpha^{k-4})$ work and $\mathcal{O}(\log^{-2} n)$ depth *w.h.p.* per update and $\mathcal{O}(m+\Delta)$ space, where $\alpha \in \mathcal{O}(\sqrt{m})$ is the arboricity of the graph. They also give a parallel fast matrix multiplication algorithm with $\mathcal{O}(\min(\Delta m^{(2k-1)\omega_p/(3\omega_p+3)}, (\Delta+m)^{2(k+1)\omega_p/(3\omega_p+3)}))$ amortized work and $\mathcal{O}(\log(\Delta+m))$ depth, with parallel matrix multiplication constant ω_p . Eppstein et al. [9] also count all three-vertex subgraphs in directed graphs in amortized time $\mathcal{O}(h)$. For specific graph classes, namely *bounded expansion* graphs, Dvorak and Tuma [8] gave a different algorithm for maintaining counts for arbitrary graph patterns \mathcal{P} of k vertices the number of induced subgraphs of pattern \mathcal{P} in amortized time $\mathcal{O}(\log^{(k^2-k)/2-1} n)$ and in amortized time $\mathcal{O}(n^\epsilon)$ for any constant $\epsilon > 0$ in *no-where dense* graphs.

In recent subsequent work [13], it was shown that counting 4-cycles is hard also in random graphs, i.e., with $\Omega(m^{1/2-\delta})$ update time or $\Omega(m^{1-\delta})$ query time.

3 New Counting Algorithms for Subgraphs on Four Vertices

Counting the number of claws [10] and 4-cliques is fairly straightforward [7]. The latter extends to 4-vertex subgraphs in general, where all work is either done during the updates or queries.

► **Observation 3** ([10]). *Let $G = (V, E)$ be a dynamic graph and \mathcal{P} be the claw \wedge . Then, $\mathbf{c}(G, \mathcal{P}) = \sum_{v \in V, \deg(v) \geq 3} \binom{\deg(v)}{3}$. The count can be initialized in $\mathcal{O}(n)$ time and maintained in constant time and space. We can query $\mathbf{c}(G, \mathcal{P}, \{u, v\})$ for an arbitrary edge $\{u, v\} \in E$ in constant time.*

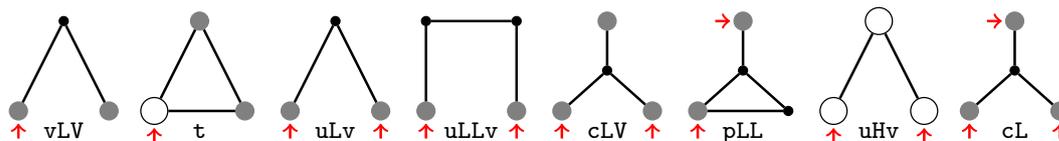
► **Theorem 4.** *Let $G = (V, E)$ be a dynamic graph and $\mathcal{P} \in \mathbb{P}_4$. We can*

- (i) *maintain $\mathbf{c}(G, \mathcal{P})$ in worst-case $\mathcal{O}(m)$ update time and constant space.*
- (ii) *query $\mathbf{c}(G, \mathcal{P}, \{u, v\})$ for an arbitrary edge $\{u, v\} \in E$ in worst-case $\mathcal{O}(m)$ time, with constant update time and space.*
- (iii) *query $\mathbf{c}(G, \mathcal{P}, \{u, v\})$ for an arbitrary edge $\{u, v\} \in E$ in worst-case constant time, with $\mathcal{O}(m)$ worst-case update time and $\mathcal{O}(m)$ space.*

For connected 4-vertex subgraphs other than the claw and the clique, we can invest space to achieve speedups in running time. In the following, we present our data structure and show how to update it efficiently. We then use different parts of the data structure to count different subgraphs. Specifically, we prove the following result:

► **Theorem 5.** *Let G be a dynamic graph and $\mathcal{P} \in \mathbb{P}_4$. We can maintain $\mathbf{c}(G, \mathcal{P})$ in*

- (i) *amortized $\mathcal{O}(\sqrt{m})$ update time with $\mathcal{O}(\min(m^{1.5}, n^2))$ space and query $\mathbf{c}(G, \mathcal{P}, e)$ for an arbitrary edge $e \in E$ in worst-case $\mathcal{O}(\sqrt{m})$ time if \mathcal{P} is the length-3 path $\not\wedge$,*
- (ii) *amortized $\mathcal{O}(m^{2/3})$ update time with $\mathcal{O}(n^2)$ space and query $\mathbf{c}(G, \mathcal{P}, e)$ for an arbitrary edge $e \in E$ in worst-case $\mathcal{O}(m^{2/3})$ time if \mathcal{P} is the paw $\not\triangleleft$ or the 4-cycle \diamond ,*
- (iii) *amortized $\mathcal{O}(m^{2/3})$ update time with $\mathcal{O}(\min(nm, m^{5/3}))$ space and query $\mathbf{c}(G, \mathcal{P}, e)$ for an arbitrary edge $e \in E$ in worst-case $\mathcal{O}(m^{2/3})$ time if \mathcal{P} is the diamond \diamond .*



■ **Figure 1** Subgraph structures of \mathcal{D}_ε . Small and filled vertices have low degree, large and empty vertices high degree, medium-sized and shaded vertices can have either high or low degree. Anchors are marked by arrows.

Our algorithm makes use of a standard technique in dynamic graph algorithms that partitions vertices into *high-degree* and *low-degree* vertices. We adapt it to our needs as follows: Let m_0 be the number of edges of G at construction or when recomputing from scratch and let $M = 2m_0$. A recomputation from scratch and re-initialization of the partition is triggered whenever the current number of edges $m < \lfloor \frac{M}{4} \rfloor$ or $m \geq M$. Initially and at each recomputation from scratch, a vertex $v \in V$ is classified as *high-degree* and added to partition \mathcal{H} if $\deg(v) \geq \theta$ and otherwise as *low-degree* and added to partition $\mathcal{L} := V \setminus \mathcal{H}$, for some threshold θ . As G evolves, a high-degree vertex v is reclassified as low and moved to \mathcal{L} only if $\deg(v) < \frac{1}{2}\theta$. Vice-versa, a low-degree vertex v is reclassified as high and moved to \mathcal{H} only if $\deg(v) \geq \frac{3}{2}\theta$. We call such a partition $(\mathcal{H}, \mathcal{L})$ a *dynamic vertex partition with threshold θ* . If $\theta = M^\varepsilon$ for some $\varepsilon \in [0, 1]$, we call the partition an ε -partition.

► **Theorem 6** (ε -Partition [15]). *Let $\varepsilon \in [0, 1]$ and consider an ε -partition $(\mathcal{H}, \mathcal{L})$ for a dynamic graph $G = (V, E)$. Then, $|\mathcal{H}| \in \mathcal{O}(m^{1-\varepsilon})$. The partition can be constructed in $\mathcal{O}(n)$ time and maintained in amortized constant time per update with amortized $\mathcal{O}(m^{-\varepsilon})$ changes to the partition per update and $\Omega(m)$ updates between two recomputations from scratch. The required space is $\mathcal{O}(n)$.*

Data Structure \mathcal{D}_ε

We assume that the algorithm can access the degree of a vertex v in constant time and, for each pair of vertices u, v determine in constant time whether $\{u, v\} \in E$. In addition, we maintain the following data structure \mathcal{D}_ε or a subset of it, if we are not only interested in counting some specific subgraphs on four vertices. All subgraph structures that are part of \mathcal{D}_ε are non-induced. See Figure 1 for visualizations.

- an ε -partition $(\mathcal{H}, \mathcal{L})$
- For each vertex $v \in V$: $\text{vLV}[v]$: the number of 2-paths $\langle \{v, x\}, \{x, y\} \rangle$ with $x \in \mathcal{L}$
- For each vertex $v \in \mathcal{H}$: $\text{t}[v]$: the number of 3-cycles $\langle \{v, x\}, \{x, y\}, \{y, v\} \rangle$
- For each distinct, unordered pair of vertices $u, v \in V$:
 - $\text{uLv}[u, v]$: the number of 2-paths $\langle \{u, x\}, \{x, v\} \rangle$ with $x \in \mathcal{L}$
 - $\text{uLLv}[u, v]$: the number of length-3 paths $\langle \{u, x\}, \{x, y\}, \{y, v\} \rangle$ with $x, y \in \mathcal{L}$
 - $\text{cLV}[u, v]$: the number of claws with a central vertex $x \in \mathcal{L}$, $u \neq x \neq v$
 - $\text{pLL}[u, v]$: the number of paws with a central vertex $x \in \mathcal{L}$, $u \neq x \neq v$, u or v at the other end of the arm, and fourth vertex $y \in \mathcal{L}$
- For each distinct, unordered pair of vertices $u, v \in \mathcal{H}$: $\text{uHv}[u, v]$: the number of length-2 paths $\langle \{v, x\}, \{x, u\} \rangle$ with $x \in \mathcal{H}$
- For each distinct, unordered triple of vertices $u, v, w \in V$: $\text{cL}[u, v, w]$: the number of claws with a fourth vertex $x \in \mathcal{L}$ at the center

18:8 Fully Dynamic Four-Vertex Subgraph Counting

For each auxiliary subgraph whose count is maintained by the data structure, we call vertices of the set that acts as key *anchors*, e.g., u and v are anchors for the 2-paths $\langle\{u, x\}, \{x, v\}\rangle$ with $x \in \mathcal{L}$, which are counted by $\mathbf{uLv}[u, v]$. We use hash tables with $\mathcal{O}(1)$ amortized access time and only store non-zero counts. Note that \mathbf{uLv} , \mathbf{uLLv} , \mathbf{pLL} , and \mathbf{cL} correspond to s_2 , s_3 , s_5 , and s_7 , respectively, in the algorithm by Eppstein et al. [9], whereas \mathbf{vLV} , \mathbf{t} , and \mathbf{cLV} are modifications of s_0 , s_1 , and s_4 , and \mathbf{uHv} has no equivalent at all. However, Eppstein et al. [9] use a different partitioning scheme, where there are at most $\mathcal{O}(h)$ vertices of degree $\Omega(h)$, whereas in our case, there are at most $\mathcal{O}(m^{1-\varepsilon})$ vertices of degree $\Omega(m^\varepsilon)$, which requires a different running time analysis also for the common auxiliary counts.

We generally assume that in case of an edge insertion, the auxiliary counts are updated immediately *before* the counts of interest, and in reverse order for an edge deletion. The update of the ε -partition can either happen first or last (but not in between). We also assume that we start with an empty graph and all counts are initialized to zero.

Maintaining the Data Structure \mathcal{D}_ε

Given a dynamic graph $G = (V, E)$ and $\varepsilon \in [0, 1]$, we show how the components of the data structure \mathcal{D}_ε can be updated after an edge insertion or deletion and if a vertex changes partition. We start with a helper lemma:

► **Lemma 7.** *Let \mathbf{aux} be an auxiliary subgraph count in \mathcal{D}_ε with worst-case update time $\mathcal{E}_{\mathbf{aux}}$ after an edge insertion or deletion, worst-case update time $\mathcal{V}_{\mathbf{aux}}$ after a vertex changes partition, and $\mathcal{S}_{\mathbf{aux}}$ space. Then, \mathcal{D}_ε with \mathbf{aux} can be maintained in amortized update time $\mathcal{O}(\mathcal{E}_{\mathbf{aux}} + \mathcal{V}_{\mathbf{aux}} \cdot m^{-\varepsilon})$ with $\mathcal{O}(n + \mathcal{S}_{\mathbf{aux}})$ space.*

Proof. By Theorem 6, the ε -partition can be maintained in $\mathcal{O}(n)$ space and such that there are $\Omega(m)$ updates between two recomputations of the partition from scratch. After each such complete repartitioning, we set \mathbf{aux} to zero and re-insert all edges one-by-one. The total recomputation time hence is $\mathcal{O}(m \cdot \mathcal{E}_{\mathbf{aux}})$ and amortization over $\Omega(m)$ edge updates results in an amortized edge update time of $\mathcal{O}(\mathcal{E}_{\mathbf{aux}})$. By Theorem 6, there are amortized $\mathcal{O}(m^{-\varepsilon})$ vertices changing partition per edge update, hence the claim follows. ◀

As the insertion and deletion operations are entirely symmetric and only differ in whether a certain amount is added or subtracted from the stored counts, we only give the details for edge insertions in the following. Similarly, we only consider the case that a vertex v changes from \mathcal{L} to \mathcal{H} ; the other case is symmetric. Note that if v is about to change partitions, $\deg(v) \in \Theta(m^\varepsilon)$.

► **Lemma 8.** *\mathcal{D}_ε with \mathbf{vLV} can be maintained in amortized $\mathcal{O}(m^\varepsilon)$ update time and $\mathcal{O}(n)$ space.*

Proof. Let $\{u, v\}$ be the newly inserted edge. If $u \in \mathcal{L}$ ($v \in \mathcal{L}$), increase $\mathbf{vLV}[w]$ by one for each $w \in N_{\bar{v}}(u)$ ($w \in N_{\bar{u}}(v)$) and increase $\mathbf{vLV}[v]$ by $\deg(u) - 1$ ($\mathbf{vLV}[u]$ by $\deg(v) - 1$). This takes $\mathcal{O}(m^\varepsilon)$ time.

If a vertex $v \in \mathcal{L}$ changes to \mathcal{H} , this affects all length-2 paths where v is the central, low-degree vertex. For each neighbor $w \in N(v)$, decrease $\mathbf{vLV}[w]$ by $\deg(v) - 1$. The running time is $\mathcal{O}(\deg(v)) = \mathcal{O}(m^\varepsilon)$.

As each vertex may be adjacent to at least one low-degree vertex, the space requirement is $\mathcal{O}(n)$. By Lemma 7, \mathcal{D}_ε with \mathbf{vLV} can hence be maintained in $\mathcal{O}(m^\varepsilon)$ amortized update time and $\mathcal{O}(n)$ space. ◀

► **Lemma 9.** \mathcal{D}_ε with \mathbf{uLv} can be maintained in amortized $\mathcal{O}(m^\varepsilon)$ time per update and $\mathcal{O}(\min(m^{1+\varepsilon}, n^2))$ space.

Proof. Let $\{u, v\}$ be the newly inserted edge. If $u \in \mathcal{L}$ ($v \in \mathcal{L}$), increment $\mathbf{uLv}[v, w]$ ($\mathbf{uLv}[u, w]$) by one for each $w \in N_{\bar{v}}(u)$ ($w \in N_{\bar{u}}(v)$). This takes $\mathcal{O}(m^\varepsilon)$ time.

If a vertex $v \in \mathcal{L}$ changes to \mathcal{H} , this affects all length-2 paths where v is the central, low-degree vertex. For each pair of distinct neighbors $x, y \in N(v)$, decrease $\mathbf{uLv}[x, y]$ by 1. The running time is $\mathcal{O}(\deg(v)^2) = \mathcal{O}(m^{2\varepsilon})$.

Each edge may be incident to at least one low-degree vertex v and form $\mathcal{O}(m^\varepsilon)$ length-2 paths with the other edges incident to v . The space requirement hence is $\mathcal{O}(\min(m^{1+\varepsilon}, n^2))$. By Lemma 7, \mathcal{D}_ε with \mathbf{uLv} can hence be maintained in $\mathcal{O}(m^\varepsilon + m^{2\varepsilon}m^{-\varepsilon}) = \mathcal{O}(m^\varepsilon)$ amortized update time and $\mathcal{O}(\min(m^{1+\varepsilon}, n^2))$ space. ◀

► **Lemma 10.** \mathcal{D}_ε with \mathbf{t} can be maintained in amortized $\mathcal{O}(m^{\max(1-\varepsilon, \varepsilon)})$ time per update and $\mathcal{O}(\min(m^{1+\varepsilon}, n^2))$ space.

Proof. Let $\{u, v\}$ be the newly inserted edge. For each $h \in \mathcal{H}$, increment $\mathbf{t}[h]$ by one if h is adjacent to both u and v . If $u \in \mathcal{H}$ ($v \in \mathcal{H}$): Increment $\mathbf{t}[u]$ ($\mathbf{t}[v]$) by one for each $h \in \mathcal{H}$ that is adjacent to both u and v , and increment $\mathbf{t}[u]$ ($\mathbf{t}[v]$) by $\mathbf{uLv}[u, v]$. This takes $\mathcal{O}(|\mathcal{H}|) = \mathcal{O}(m^{1-\varepsilon})$ time.

If a vertex $v \in \mathcal{L}$ changes to \mathcal{H} , then for each pair of distinct neighbors $x, y \in N(v)$ such that $\{x, y\} \in E$, we increase $\mathbf{t}[v]$ by one. Otherwise, if v changes from \mathcal{H} to \mathcal{L} , set $\mathbf{t}[v] := 0$. The running time is $\mathcal{O}(\deg(v)^2) = \mathcal{O}(m^{2\varepsilon})$.

By Lemma 9, \mathcal{D}_ε with \mathbf{uLv} can be maintained in amortized $\mathcal{O}(m^\varepsilon)$ update time and $\mathcal{O}(\min(m^{1+\varepsilon}, n^2))$ space. As $|\mathcal{H}| \in \mathcal{O}(m^{1-\varepsilon})$, the space requirement for \mathbf{t} is $\mathcal{O}(\min(m^{1+\varepsilon}, n^2))$. By Lemma 7, \mathcal{D}_ε with \mathbf{t} can be maintained in $\mathcal{O}(m^{1-\varepsilon} + m^{2\varepsilon}m^{-\varepsilon}) + \mathcal{O}(m^\varepsilon) = \mathcal{O}(m^{\max(1-\varepsilon, \varepsilon)})$ amortized update time and $\mathcal{O}(\min(m^{1+\varepsilon}, n^2))$ space. ◀

► **Lemma 11.** \mathcal{D}_ε with \mathbf{uLLv} can be maintained in amortized $\mathcal{O}(m^{2\varepsilon})$ time per update and $\mathcal{O}(\min(m^{1+2\varepsilon}, n^2))$ space.

Proof. Let $\{u, v\}$ be the newly inserted edge. If $u \in \mathcal{L}$ ($v \in \mathcal{L}$), we count the length-3 paths starting/ending with $\{u, v\}$ as follows: For each low-degree neighbor $w \in N_{\bar{v}}(u) \cap \mathcal{L}$ ($w \in N_{\bar{u}}(v) \cap \mathcal{L}$), increment $\mathbf{uLLv}[v, x]$ ($\mathbf{uLLv}[u, x]$) by one for each $x \in N(w) \setminus \{u, v\}$. This takes $\mathcal{O}(m^{2\varepsilon})$ time. If both $u \in \mathcal{L}$ and $v \in \mathcal{L}$, we additionally count the length-3 paths having $\{u, v\}$ as centerpiece in $\mathcal{O}(\deg(u)^2) = \mathcal{O}(m^{2\varepsilon})$ time: For each pair of distinct vertices x, y with $x \in N_{\bar{v}}(u)$, $y \in N_{\bar{u}}(v)$, increment $\mathbf{uLLv}[x, y]$ by one.

If a vertex $v \in \mathcal{L}$ changes to \mathcal{H} , we iterate over all pairs of distinct vertices y, w , where $y \in N_{\bar{v}}(x)$ for some low-degree neighbor $x \in N(v) \cap \mathcal{L}$ and $w \in N_{\bar{x}}(v)$, and decrease $\mathbf{uLLv}[w, y]$ by one. As v has $\mathcal{O}(m^{2\varepsilon})$ pairs of neighbors and each low-degree neighbor has in turn $\mathcal{O}(m^\varepsilon)$ neighbors, the running time is in $\mathcal{O}(m^{3\varepsilon})$.

Each edge may be incident to two low-degree vertices and hence form $\mathcal{O}(m^{2\varepsilon})$ length-3 paths with the other edges incident to the end vertices. The space requirement hence is $\mathcal{O}(\min(m^{1+2\varepsilon}, n^2))$. By Lemma 7, \mathcal{D}_ε with \mathbf{uLLv} can be maintained in $\mathcal{O}(m^{2\varepsilon} + m^{3\varepsilon}m^{-\varepsilon}) = \mathcal{O}(m^{2\varepsilon})$ amortized update time and $\mathcal{O}(\min(m^{1+2\varepsilon}, n^2))$ space. ◀

► **Lemma 12.** \mathcal{D}_ε with \mathbf{cLv} can be maintained in amortized $\mathcal{O}(m^{2\varepsilon})$ time per update and $\mathcal{O}(n \min(n, m^{2\varepsilon}))$ space.

Proof. Let $\{u, v\}$ be the newly inserted edge. If $u \in \mathcal{L}$ ($v \in \mathcal{L}$), we update the number of claws where u (v) is the central vertex as follows: For each pair of distinct neighbors $x, y \in N_{\bar{v}}(u)$ ($x, y \in N_{\bar{u}}(v)$), increment $\mathbf{cLv}[x, y]$ by one. This accommodates for the claws

18:10 Fully Dynamic Four-Vertex Subgraph Counting

where $v(u)$ is not an anchor vertex and takes $\mathcal{O}(m^{2\varepsilon})$ time. For the other case, increment $\text{cLV}[v, w]$ ($\text{cLV}[u, w]$) by $\deg(u) - 2$ ($\deg(v) - 2$) for each $w \in N_{\bar{v}}(u)$ ($w \in N_{\bar{u}}(v)$) in $\mathcal{O}(m^\varepsilon)$ time.

If a vertex $v \in \mathcal{L}$ changes to \mathcal{H} , we decrease $\text{cLV}[x, y]$ by $\deg(v) - 2$ for each pair of distinct neighbors $x, y \in N(v)$ in total $\mathcal{O}(\deg(v)^2) = \mathcal{O}(m^{2\varepsilon})$ time.

As each vertex may be adjacent to at least one low-degree vertex and we store the count for all pairs, the space requirement is in $\mathcal{O}(n^2)$. On the other hand, each low-degree vertex has at most $\mathcal{O}(m^{2\varepsilon})$ neighbors that can serve as anchors, which yields a space requirement of $\mathcal{O}(nm^{2\varepsilon})$. By Lemma 7, \mathcal{D}_ε with cLV can be maintained in $\mathcal{O}(m^{2\varepsilon} + m^{2\varepsilon}m^{-\varepsilon}) = \mathcal{O}(m^{2\varepsilon})$ amortized update time and $\mathcal{O}(n \min(n, m^{2\varepsilon}))$ space. \blacktriangleleft

► **Lemma 13.** \mathcal{D}_ε with pLL can be maintained in amortized $\mathcal{O}(m^{2\varepsilon})$ time per update and $\mathcal{O}(\min(n^2, nm^{2\varepsilon}))$ space.

Proof. Let $\{u, v\}$ be the newly inserted edge.

If $u \in \mathcal{L}$ ($v \in \mathcal{L}$): First, we update all paws where $u(v)$ is the central vertex and $v(u)$ is the anchor vertex at the arm: For each ordered pair of distinct neighbors $x, y \in N_{\bar{v}}(u)$ ($x, y \in N_{\bar{u}}(v)$) such that $x \in \mathcal{L}$ and $\{x, y\} \in E$, increment $\text{pLL}[v, y]$ ($\text{pLL}[u, y]$) by one. This can be done in $\mathcal{O}(m^{2\varepsilon})$ time. Second, we update all paws where $u(v)$ is the central vertex and $v(u)$ is the anchor vertex in the triangle: For each ordered pair of distinct neighbors $x, y \in N_{\bar{v}}(u)$ ($x, y \in N_{\bar{u}}(v)$) such that $x \in \mathcal{L}$ and $\{x, v\} \in E$ ($\{x, u\} \in E$), increment $\text{pLL}[v, y]$ ($\text{pLL}[u, y]$) by one. This again can be done in $\mathcal{O}(m^{2\varepsilon})$ time. Third, we update all paws where $u(v)$ is the non-anchor, non-central vertex in the triangle and $v(u)$ is the anchor vertex in the triangle: For each neighbor $x \in N_{\bar{v}}(u)$ ($x \in N_{\bar{u}}(v)$) with $x \in \mathcal{L}$ and $\{v, x\} \in E$ ($\{u, x\} \in E$), increment $\text{pLL}[v, y]$ ($\text{pLL}[u, y]$) by one for each $y \in N(x) \setminus \{u, v\}$. The running time is in $\mathcal{O}(m^{2\varepsilon})$, as $u, x \in \mathcal{L}$.

If both $u \in \mathcal{L}$ and $v \in \mathcal{L}$, we update all paws where $\{u, v\}$ connects the central vertex to the non-anchor vertex in the triangle: For each ordered pair of distinct neighbors $x, y \in N_{\bar{v}}(u)$ with $\{x, v\} \in E$ and each ordered pair of distinct neighbors $x, y \in N_{\bar{u}}(v)$ with $\{x, u\} \in E$, increment $\text{pLL}[x, y]$ by one. The running time is in $\mathcal{O}(\deg(u)^2 + \deg(v)^2) = \mathcal{O}(m^{2\varepsilon})$.

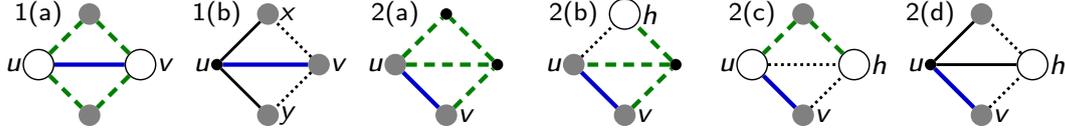
If a vertex $v \in \mathcal{L}$ changes to \mathcal{H} : For all paws where v was the central vertex, we iterate over all unordered pairs of neighbors $y, z \in N(v)$ and every neighbor $x \in N(v) \cap \mathcal{L}$ such that $y \neq x \neq z$. If $\{x, y\} \in E$, we decrease $\text{pLL}[y, z]$ by one, and if $\{x, z\} \in E$, we also decrease $\text{pLL}[y, z]$ by one. For all paws where v was the low-degree, non-central vertex in the triangle, we iterate over all pairs of distinct neighbors $x, y \in N(v)$ such that $\{x, y\} \in E$ and $x \in \mathcal{L}$, and, for each $z \in N(x) \setminus \{v, y\}$, decrease $\text{pLL}[y, z]$ by one. In this case, $\{x, z\}$ forms the arm. As $\deg(x) \in \mathcal{O}(m^\varepsilon)$ in the second case, the total running time is $\mathcal{O}(\deg(v)^3 + \deg(v)^2 \cdot m^\varepsilon) = \mathcal{O}(m^{3\varepsilon})$.

The argument for the space requirement is the same as for to cLV and $\mathcal{O}(n \min(n, m^{2\varepsilon}))$ by Lemma 12. By Lemma 7, \mathcal{D}_ε with pLL can be maintained in $\mathcal{O}(m^{2\varepsilon} + m^{3\varepsilon}m^{-\varepsilon}) = \mathcal{O}(m^{2\varepsilon})$ amortized update time and $\mathcal{O}(n \min(n, m^{2\varepsilon}))$ space. \blacktriangleleft

► **Lemma 14.** \mathcal{D}_ε with uHv can be maintained in amortized $\mathcal{O}(m^{\max(1-\varepsilon, \varepsilon)})$ time per update and $\mathcal{O}(n + \min(n^2, m^{2-2\varepsilon}))$ space.

Proof. Let $\{u, v\}$ be the newly inserted edge. If $u, v \in \mathcal{H}$, we iterate over all $h \in \mathcal{H} \setminus \{u, v\}$. If h is adjacent to $u(v)$, increment $\text{uHv}[h, v]$ ($\text{uHv}[u, h]$), respectively, by one. The running time is $\mathcal{O}(|\mathcal{H}|) = \mathcal{O}(m^{1-\varepsilon})$.

If a vertex $v \in \mathcal{L}$ changes to \mathcal{H} (analogously vice-versa): For each pair of distinct high-degree neighbors $x, y \in N(v) \cap \mathcal{H}$, increment $\text{uHv}[x, y]$ by one in total $\mathcal{O}(\deg(v)^2) = \mathcal{O}(m^{2\varepsilon})$ time. *Only if v changes from \mathcal{L} to \mathcal{H} :* For every high-degree neighbor $w \in N(v) \cap \mathcal{H}$, we



■ **Figure 2** Counting the number of diamonds that contain an edge $\{u, v\}$. Green, dashed edges belong to paths that are considered via auxiliary counts, whereas dotted edges are edges whose presence is looked up by the algorithm. As before, small and filled vertices have low degree, large and empty vertices high degree, medium-sized and shaded vertices can have either high or low degree.

iterate over all $h \in \mathcal{H}$ and increase $\text{uHv}[v, h]$ by one if $\{w, h\} \in E$ in total $\mathcal{O}(\deg(v) \cdot |\mathcal{H}|) = \mathcal{O}(m^\varepsilon \cdot m^{1-\varepsilon}) = \mathcal{O}(m)$ time. Only if v changes from \mathcal{H} to \mathcal{L} , we set $\text{uHv}[v, h] := 0$ for each $h \in \mathcal{H}$ in total $\mathcal{O}(|\mathcal{H}|) = \mathcal{O}(m^{1-\varepsilon})$ time. The overall time is hence $\mathcal{O}(m^{\max(2\varepsilon, 1)})$.

There are $\mathcal{O}(\min(n, m^{1-\varepsilon}))$ high-degree vertices, which results in $\mathcal{O}(\min(n^2, m^{2-2\varepsilon}))$ pairs of anchor vertices. \mathcal{D}_ε with uHv can hence be maintained in $\mathcal{O}(m^{1-\varepsilon} + m^{\max(2\varepsilon, 1)}m^{-\varepsilon}) = \mathcal{O}(m^{\max(1-\varepsilon, \varepsilon)})$ amortized update time and $\mathcal{O}(n + \min(n^2, m^{2-2\varepsilon}))$ space by Lemma 7. ◀

► **Lemma 15.** \mathcal{D}_ε with cL can be maintained in amortized $\mathcal{O}(m^{2\varepsilon})$ time per update and $\mathcal{O}(\min(n^3, nm^{3\varepsilon}, m^{1+2\varepsilon}))$ space.

Proof. Let $\{u, v\}$ be the newly inserted edge. If $u \in \mathcal{L}$ ($v \in \mathcal{L}$), increment $cL[v, x, y]$ ($cL[u, x, y]$) by one for each pair of distinct neighbors $x, y \in N_{\bar{v}}(u)$ ($x, y \in N_{\bar{u}}(v)$). This takes $\mathcal{O}(m^{2\varepsilon})$ time.

If a vertex $v \in \mathcal{L}$ changes to \mathcal{H} : For each triple of distinct neighbors $x, y, z \in N(v)$, we decrease $cL[x, y, z]$ by one in total $\mathcal{O}(\deg(v)^3) = \mathcal{O}(m^{3\varepsilon})$ time.

As each edge may be incident to a low-degree vertex v , the number of triples with non-zero count for cL is in $\mathcal{O}(\min(n^3, nm^{3\varepsilon}, m^{1+2\varepsilon}))$. By Lemma 7, \mathcal{D}_ε with cL can be maintained in $\mathcal{O}(m^{2\varepsilon} + m^{3\varepsilon-\varepsilon}) = \mathcal{O}(m^{2\varepsilon})$ amortized update time and $\mathcal{O}(\min(n^3, m^{1+2\varepsilon}))$ space. ◀

Non-Induced Subgraph Counts

We are now ready to prove Theorem 5 and show for each connected subgraph on four vertices how to count it using the data structure \mathcal{D}_ε .

► **Lemma 16.** Let $G = (V, E)$ be a dynamic graph, $\varepsilon \in [0, 1]$, and \mathcal{P} be the diamond \diamond . We can query $c(G, \mathcal{P}, \{u, v\})$ for an arbitrary edge $\{u, v\} \in E$ in $\mathcal{O}(\min(m^{\max(1-\varepsilon, 2\varepsilon)}, n^2))$ worst-case time if we maintain the data structure \mathcal{D}_ε with auxiliary counts uLv , pLL , uHv , and cL .

Proof. Edge $\{u, v\}$ can either be the chord of the diamond or be part of the 4-cycle. See Figure 2 for an illustration.

For the first case, where $\{u, v\}$ is the chord: (a) If $u, v \in \mathcal{H}$, we can obtain the number of length-2 paths p between u and v as $p = \text{uLv}[u, v] + \text{uHv}[u, v]$. As each pair of length-2 paths forms a diamond with $\{u, v\}$, the total number of diamonds is $\binom{p}{2}$. (b) Otherwise, $\{u, v\} \cap \mathcal{L} \neq \emptyset$. W.l.o.g., $u \in \mathcal{L}$. We then iterate over all distinct, unordered pairs of neighbors $x, y \in N_{\bar{v}}(u)$ in $\mathcal{O}(\deg(u)^2) = \mathcal{O}(\min(m^{2\varepsilon}, n^2))$ time. For each such pair with $\{x, v\}, \{y, v\} \in E$, we count one diamond.

For the second case, where $\{u, v\}$ is part of the cycle, we distinguish between the degrees of the other two vertices. (a) The number of diamonds where the other two vertices have low degree is given by $\text{pLL}[u, v]$. Note that either u or v is incident to the chord. (b) The number of diamonds where the other vertex incident to the chord has low degree and

18:12 Fully Dynamic Four-Vertex Subgraph Counting

the fourth vertex has high degree can be obtained by iterating over all $h \in \mathcal{H} \setminus \{u, v\}$ in $\mathcal{O}(|\mathcal{H}|) = \mathcal{O}(\min(m^{1-\varepsilon}, n))$ time. If either $\{h, u\} \in E$ or $\{h, v\} \in E$, we have $\text{cL}[u, v, h]$ more diamonds. If both $\{h, u\}, \{h, v\} \in E$, we add $2\text{cL}[u, v, h]$ to the number of diamonds. (c, d) The number of diamonds where the other vertex incident to the chord has high degree can be obtained as follows: (c) If $u \in \mathcal{H}$ ($v \in \mathcal{H}$), the number of diamonds where the chord is incident to u (v) can be obtained by iterating over all $h \in \mathcal{H} \setminus \{u, v\}$ in $\mathcal{O}(|\mathcal{H}|) = \mathcal{O}(\min(m^{1-\varepsilon}, n))$ time. For each such vertex h , we check whether $\{u, h\}, \{v, h\} \in E$ and add $\text{uLv}[u, h] + \text{uHv}[u, h] - 1$ ($\text{uLv}[v, h] + \text{uHv}[v, h] - 1$) to the count. The correction by 1 is necessary because the auxiliary counts also contain the path $\langle \{u, v\}, \{v, h\} \rangle$ ($\langle \{h, u\}, \{u, v\} \rangle$). (d) If $u \in \mathcal{L}$ ($v \in \mathcal{L}$), we iterate over all high-degree neighbors $h \in N_{\bar{v}}(u) \cap \mathcal{H}$ ($h \in N_{\bar{u}}(v) \cap \mathcal{H}$) and in each case over all $x \in N(u) \setminus \{v, h\}$ ($x \in N(v) \setminus \{u, h\}$) in total $\mathcal{O}(\min(m^{2\varepsilon}, n^2))$ time and count one diamond each if $\{h, x\}, \{h, v\} \in E$ ($\{h, x\}, \{h, u\} \in E$). ◀

► **Lemma 17.** *Let $G = (V, E)$ be a dynamic graph and \mathcal{P} be the diamond \diamond . We can maintain $\mathbf{c}(G, \mathcal{P})$ in amortized $\mathcal{O}(m^{2/3})$ update time and $\mathcal{O}(\min(nm, m^{5/3}))$ space. We can query $\mathbf{c}(G, \mathcal{P}, e)$ for an arbitrary edge $e \in E$ in worst-case $\mathcal{O}(m^{2/3})$ time.*

Proof. After an edge $\{u, v\}$ was inserted or before an edge $\{u, v\}$ is removed, the number of diamonds containing it can be obtained in $\mathcal{O}(\min(m^{\max(1-\varepsilon, 2\varepsilon)}, n^2))$ time worst-case time by Lemma 16 if \mathcal{D}_ε with auxiliary counts uLv , pLL , uHv , and cL is maintained. By Lemma 9, Lemma 13, Lemma 14, and Lemma 15, this can be done in amortized $\mathcal{O}(m^\varepsilon + m^{\max(1-\varepsilon, 2\varepsilon)} + m^{2\varepsilon}) = \mathcal{O}(m^{\max(1-\varepsilon, 2\varepsilon)})$ time and $\mathcal{O}(\min(n^3, \max(n^2, nm^{3\varepsilon}, m^{2-2\varepsilon}, m^{1+2\varepsilon})))$ space. Together with the cost for the query, this yields a total amortized update time of $\mathcal{O}(m^{\max(1-\varepsilon, 2\varepsilon)}) = \mathcal{O}(m^{2/3})$ for $\varepsilon = \frac{1}{3}$ and $\mathcal{O}(\min(nm, m^{5/3}))$ space. By Lemma 16, the worst-case time to query $\mathbf{c}(G, \mathcal{P}, e)$ for an arbitrary edge $e \in E$ then is $\mathcal{O}(m^{2/3})$. ◀

Queries with Vertices and Edges and Non-Induced s -Subgraph Counts

With similar techniques, we can count non-induced triangles containing a specified vertex or edge as well as maintain s -subgraph counts for patterns with up to four vertices.

► **Theorem 18.** *Let $G = (V, E)$ be a dynamic graph, \mathcal{P} be the 3-cycle \triangle , and $\varepsilon \in [0, 1]$. We can query $\mathbf{c}(G, \mathcal{P}, a)$ for an arbitrary vertex or edge a in*

- (i) *worst-case $\mathcal{O}(\min(m^{2\varepsilon}, n^2))$ time with $\mathcal{O}(m^{\max(\varepsilon, 1-\varepsilon)})$ amortized update time and $\mathcal{O}(\min(n^2, m^{1+\varepsilon}))$ space if $a \in V$,*
- (ii) *worst-case time $\mathcal{O}(\min(m^{1-\varepsilon}, n))$ with $\mathcal{O}(m^\varepsilon)$ amortized update time and $\mathcal{O}(\min(n^2, m^{1+\varepsilon}))$ space if $a \in E$.*

► **Corollary 19.** *Let $G = (V, E)$ be a dynamic graph and \mathcal{P} the 3-cycle \triangle . We can maintain $\mathbf{c}(G, \mathcal{P})$ with an amortized update time of $\mathcal{O}(\sqrt{m})$ and $\mathcal{O}(\min(n^2, m^{1.5}))$ space and query $\mathbf{c}(G, \mathcal{P}, e)$ for $e \in E$ arbitrary in worst-case $\mathcal{O}(\sqrt{m})$ time. We can query $\mathbf{c}(G, \mathcal{P}, v)$ for arbitrary $v \in V$ in worst-case $\mathcal{O}(m^{2/3})$ time with an amortized update time of $\mathcal{O}(m^{2/3})$ and $\mathcal{O}(\min(n^2, m^{4/3}))$ space.*

► **Theorem 20.** *Let $G = (V, E)$ be a dynamic graph, $s \in V$, and \mathcal{P} be a connected subgraph. We can maintain the non-induced s -subgraph count $\mathbf{c}(G, \mathcal{P}, s)$ in*

- (i) *worst-case constant update time and constant space if \mathcal{P} is the claw λ ,*
- (ii) *amortized update time $\mathcal{O}(\sqrt{m})$ and $\mathcal{O}(n)$ space if \mathcal{P} is the 3-cycle \triangle or the length-3 path ℓ' ,*
- (iii) *amortized update time $\mathcal{O}(m^{2/3})$ and $\mathcal{O}(n^2)$ space if \mathcal{P} is the paw \blacktriangleleft , the 4-cycle \diamond , or the diamond \diamond ,*
- (iv) *worst-case update time $\mathcal{O}(m)$ and constant space if \mathcal{P} is the 4-clique \boxtimes .*

4 Lower Bounds

We give new lower bounds for detecting and counting induced and non-induced subgraphs.

Induced Subgraph Counts

Our results for counting induced subgraphs on four vertices are conditioned on the combinatorial k -clique conjecture:

► **Theorem 21.** *Let G be a dynamic graph, $\mathcal{P} \in \mathbb{P}_4$, and let $\gamma > 0$ be a small constant. There is no incremental or fully dynamic combinatorial algorithm with preprocessing time $\mathcal{O}(m^{2-\gamma})$ for maintaining $\mathbf{c}_I(G, \mathcal{P})$ in amortized update time $\mathcal{O}(m^{1-\gamma})$ and query time $\mathcal{O}(m^{2-\gamma})$, unless the k -clique conjecture fails.*

Before we turn to the proof, we recall the following relations between subgraph counts.

► **Lemma 22** ([9]). *For each pair $P, P' \in \mathbb{P}_4$, the non-induced subgraph count $\mathbf{c}(P, P') = 1$ if $P = P'$ and otherwise nonzero only in the following cases:*

$$\begin{array}{llll} \mathbf{c}(\neg\triangleleft, \neg\triangleleft) = 2, & \mathbf{c}(\diamond, \neg\triangleleft) = 4, & \mathbf{c}(\diamond, \triangleleft) = 6, & \mathbf{c}(\boxtimes, \neg\triangleleft) = 12, \\ \mathbf{c}(\neg\triangleleft, \wedge) = 1, & \mathbf{c}(\diamond, \wedge) = 2, & \mathbf{c}(\boxtimes, \wedge) = 4, & \mathbf{c}(\diamond, \neg\triangleleft) = 4, \\ \mathbf{c}(\boxtimes, \neg\triangleleft) = 12, & \mathbf{c}(\diamond, \diamond) = 1, & \mathbf{c}(\boxtimes, \diamond) = 3, & \mathbf{c}(\boxtimes, \diamond) = 6. \end{array}$$

► **Proposition 23** ([17]). *Let G, \mathcal{P} be graphs and let k be the number of vertices of \mathcal{P} . Then, $\mathbf{c}(G, \mathcal{P}) = \sum_{P \in \mathbb{P}_k} \mathbf{c}_I(G, P) \cdot \mathbf{c}(P, \mathcal{P})$.*

► **Lemma 24.** *Let $G = (V, E)$ be a graph. The following relationships between induced and non-induced subgraph counts hold:*

$$\begin{array}{l} \mathbf{c}_I(G, \boxtimes) = \mathbf{c}(G, \boxtimes) \\ \mathbf{c}_I(G, \diamond) = \mathbf{c}(G, \diamond) - 6\mathbf{c}_I(G, \boxtimes) \\ \mathbf{c}_I(G, \triangleleft) = \mathbf{c}(G, \triangleleft) - \mathbf{c}(G, \diamond) + 3\mathbf{c}_I(G, \boxtimes) \\ \mathbf{c}_I(G, \neg\triangleleft) = \mathbf{c}(G, \neg\triangleleft) - 4\mathbf{c}(G, \diamond) + 12\mathbf{c}_I(G, \boxtimes) \\ \mathbf{c}_I(G, \wedge) = \mathbf{c}(G, \wedge) - \mathbf{c}(G, \neg\triangleleft) + 2\mathbf{c}(G, \diamond) - 4\mathbf{c}_I(G, \boxtimes) \\ \mathbf{c}_I(G, \neg\triangleleft) = \mathbf{c}(G, \neg\triangleleft) - 2\mathbf{c}(G, \diamond) - 4\mathbf{c}(G, \triangleleft) + 6\mathbf{c}(G, \diamond) - 12\mathbf{c}_I(G, \boxtimes) \end{array}$$

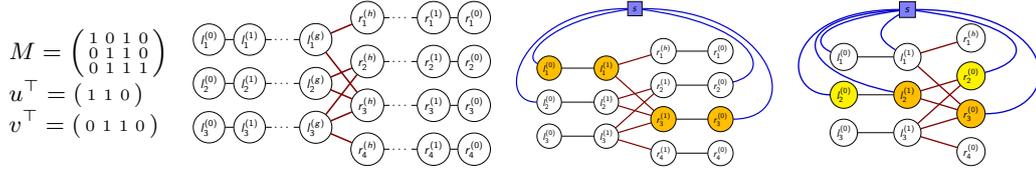
Proof. Let $\mathcal{P} \in \mathbb{P}_4$. The statement follows from Proposition 23 and Lemma 22 by substituting equations and solving them for $\mathbf{c}_I(G, \mathcal{P})$ in the order as listed. ◀

► **Corollary 25.** *Let $G = (V, E)$ be a dynamic graph and $\mathcal{P} \in \mathbb{P}_4$. We can maintain $\mathbf{c}_I(G, \mathcal{P})$ with $\mathcal{O}(m)$ worst-case update time and constant space.*

Proof of Theorem 21. First consider the case that $\mathcal{P} = \boxtimes$. Suppose there is an incremental or fully dynamic algorithm \mathcal{A} that maintains $\mathbf{c}_I(G, \mathcal{P})$ in time $\mathcal{O}(m^{1-\gamma})$ with query time $\mathcal{O}(m^{2-\gamma})$ for some $\gamma > 0$. Construct an algorithm \mathcal{A}' for static 4-clique detection as follows: Run \mathcal{A} on an initially empty graph, insert all edges one-by-one in total $\mathcal{O}(m^{2-\gamma})$ time, and query the result in $\mathcal{O}(m^{2-\gamma})$ time. As $\mathcal{O}(m^{2-\gamma}) = \mathcal{O}(n^{4-2\gamma})$ this contradicts Conjecture 2.

For the remaining five induced four-vertex subgraphs, let \mathcal{P} be such a subgraph. We construct a deterministic algorithm \mathcal{A}' for static 4-clique detection as follows: \mathcal{A}' executes the above operations for our non-induced subgraph counting algorithm from Sect. 3, which can maintain the number of all connected subgraphs on four vertices with $\mathcal{O}(m^{2/3})$ amortized update time and $\mathcal{O}(1)$ query time by Theorem 5. It thus takes $\mathcal{O}(m^{5/3})$ time in total to compute $\mathbf{c}(G, \mathcal{P})$ for all $\mathcal{P}' \in \mathbb{P}_4 \setminus \{\boxtimes\}$. Assume by contradiction that there exists an algorithm

18:14 Fully Dynamic Four-Vertex Subgraph Counting



■ **Figure 3** Construction of $G_{M,g,h}$ and G for $u^\top M v$ for (s) -5-cycle and (s) -diamond detection.

\mathcal{A}^* that maintains $\mathbf{c}_I(G, \mathcal{P})$ in update time $\mathcal{O}(m^{1-\gamma})$ and query time $\mathcal{O}(m^{2-\gamma})$. Then \mathcal{A}' also executes the same operations with \mathcal{A}^* to compute $\mathbf{c}_I(G, \mathcal{P})$. Using the formula for \mathcal{P} in Lemma 24, \mathcal{A}' can solve the static 4-clique detection problem in time $\mathcal{O}(m^{\max(5/3, 2-\gamma)}) \subseteq \mathcal{O}(n^{4-\delta})$ time for $\delta = \min(2\gamma, 2/3) > 0$, a contradiction to Conjecture 2. ◀

Non-Induced Subgraph Counts

In this section, we give new lower bounds for detecting (and thus counting) cycles of arbitrary length, paws, diamonds, and 4-cliques¹, as well as counting length-3 paths. Our results are based on the OMv conjecture. In [12] it is proven that instead of reducing from OMv directly it suffices to reduce from the following 1-uMv version: For any positive integer parameters n_1, n_2 , given an $n_1 \times n_2$ matrix M , there is no algorithm with preprocessing time polynomial in n_1 and n_2 that computes for an n_1 -dimensional vector u and an n_2 -dimensional vector v the product $u^\top M v$ in time $\mathcal{O}(n_1 n_2^{1-\delta} + n_1^{1-\delta} n_2)$ for any small constant $\delta > 0$ with error probability of at most $\frac{1}{3}$ in the word-RAM model with $\mathcal{O}(\log n)$ bit words.

► **Theorem 26.** *Let G be a partially dynamic graph and let \mathcal{P} be the paw \triangleleft , the diamond \diamond , the 4-clique \boxtimes , or a k -cycle with $k \geq 3$. On condition of Conjecture 1, there is no partially dynamic algorithm to maintain whether $\mathbf{c}(G, \mathcal{P}) > 0$ with polynomial preprocessing time and worst-case update time $\mathcal{O}(m^{1/2-\delta})$ and query time $\mathcal{O}(m^{1-\delta})$ with an error probability of at most $1/3$ for any $\delta > 0$. This also holds for fully dynamic algorithms with amortized update time and for paws, diamonds, 4-cliques, or odd k -cycles containing a specific vertex s , as well as for maintaining the number of length-3 paths $\not\sim$ and the number of length-3 paths containing a specific vertex s .*

Our constructions build on the following graph (see Figure 3 for an example).

► **Definition 27** ($G_{M,g,h}$). *Given a matrix $M \in \{0, 1\}^{n_1 \times n_2}$ and two integers $g, h \geq 0$, we denote by $G_{M,g,h} = (\bigcup_{0 \leq p \leq g} L^{(p)} \cup \bigcup_{0 \leq q \leq h} R^{(q)}, E_L \cup E_R \cup E_M)$ the $(g+h+2)$ -partite graph with*

$$L^{(p)} = \{l_1^{(p)}, \dots, l_{n_1}^{(p)}\}, \quad 0 \leq p \leq g; \quad E_L = \{(l_i^{(p)}, l_i^{(p+1)}) \mid 1 \leq i \leq n_1 \wedge 0 \leq p < g\}$$

$$R^{(q)} = \{r_1^{(q)}, \dots, r_{n_2}^{(q)}\}, \quad 0 \leq q \leq h; \quad E_R = \{(r_j^{(q)}, r_j^{(q+1)}) \mid 1 \leq j \leq n_2 \wedge 0 \leq q < h\}$$

and $E_M = \{(l_i^{(g)}, r_j^{(h)}) \mid M_{ij} = 1\}$. $G_{M,g,h}$ has $(g+1) \cdot n_1 + (h+1) \cdot n_2$ vertices and at most $n_1 n_2 + g \cdot n_1 + h \cdot n_2$ edges. All vertices x in $L^{(p)}$ for $1 \leq p < g$ and in $R^{(q)}$ for $1 \leq q < h$ have $\deg(x) = 2$, whereas all vertices y in $L^{(0)}$ and in $R^{(0)}$ have $\deg(y) = 1$.

For convenience, we set $L := L^{(0)}$, $R := R^{(0)}$, $l_i := l_i^{(0)}$, and $r_j := r_j^{(0)}$.

► **Observation 28.** *Every cycle in $G_{M,g,h}$ is even and has length at least 4.*

¹ Theorem 21 applies also to 4-cliques, but it is based on a different assumption.

Let s be a fixed vertex in the graph. The s - k -cycle detection problem requires the algorithm to detect whether a k -cycle containing s exists. We use the same notation for the other subgraphs as well.

► **Lemma 29.** *Given a partially dynamic algorithm \mathcal{A} for one of the problems listed below, one can solve 1-uMv with parameters n_1 and n_2 by running the preprocessing step of \mathcal{A} on a graph with $\mathcal{O}(m + \sqrt{m} \cdot k)$ edges and $\Theta(\sqrt{m} \cdot k)$ vertices, and then making $\mathcal{O}(\sqrt{m})$ insertions (or $\mathcal{O}(\sqrt{m})$ deletions) and 1 query, where m is such that $n_1 = n_2 = \sqrt{m}$. The problems are*

- | | |
|--|--|
| (a) (s) - k -Cycle Detection for odd k | (d) (s) - k -Clique Detection for $k = 4$ |
| (b) (s) -Paw Detection | (e) (s) -length- k Path Counting for $k = 3$ |
| (c) (s) -Diamond Detection | (f) k -Cycle Detection |

Proof of Case (c). We only prove the decremental case. Consider a 1-uMv problem with $n_1 = n_2 = \sqrt{m}$. Given M , we construct the tripartite graph G from $G_{M,1,0}$ by adding to it a vertex s and connecting it by an edge to every vertex in $G_{M,1,0}$. Thus, the total number of edges is at most $n_1 n_2 + 3n_1 + n_2 = \mathcal{O}(m)$. Once u and v arrive, we delete $\{s, l_i\}$ and $\{s, l_i^{(1)}\}$ iff $u_i = 0$ and delete $\{r_j, s\}$ iff $v_j = 0$. See Figure 3 for an example.

Consider the case that G contains a diamond with chord e . As every triangle must be incident to s by Observation 28, $e = \{s, x\}$ for some $x \in L \cup L^{(1)} \cup R$. Furthermore, all vertices in L have degree at most two and x has degree at least three, so $x \notin L$. If $x = r_j \in R$, then by construction, x must have two neighbors $l_i^{(1)}, l_{i'}^{(1)} \in L^{(1)}$ such that there are edges $\{s, l_i^{(1)}\}$ and $\{s, l_{i'}^{(1)}\}$ in G . Again by construction, there are thus also edges $\{s, l_i\}$ and $\{s, l_{i'}\}$ and a diamond $\{s, l_i, l_{i'}^{(1)}, x = r_j\}$ with chord $\{s, l_i^{(1)}\}$. As each vertex in $L^{(1)}$ is adjacent to exactly one vertex in L , every diamond must contain a vertex $r \in R$ and the edge $\{r, s\}$. Hence, we have $u^\top M v = 1$ iff there is a diamond in G iff there is a diamond incident to s . In total, we need to do $2n_1 + n_2 = \mathcal{O}(\sqrt{m})$ updates and 1 query. ◀

5 Conclusion

Our focus in this work was especially on non-induced and induced four-vertex subgraphs. We gave improved both upper and lower bounds for detecting or counting four-vertex subgraphs in the dynamic setting, thereby closing the gap (w.r.t. improvements by a polynomial factor) for counting non-induced length-3 paths and narrowing it considerably for non-induced paws, 4-cycles, and diamonds. For counting induced subgraphs, we showed that the update time of the algorithm by Eppstein et al. [9] cannot be improved by a polynomial factor, but that a better space complexity can be achieved in the worst case.

Many of our lower bounds also apply to subgraphs with more than four vertices, but to the best of our knowledge, only algorithms for cliques have been considered here so far. Hence, besides closing the gap for four-vertex subgraphs, the complexity of detecting and counting subgraphs with five or more vertices would be an interesting field for future work.

We also investigated the complexity of querying the number of subgraphs containing a specific edge, as such queries are relevant, e.g., to measure the similarity of graphs via histograms. This can similarly be done for vertices. As a by-product of our results for four-vertex subgraphs, we showed for 3-cycles that such vertex queries can be answered in $\mathcal{O}(m^{\frac{2}{3}})$. The complexity of vertex queries for larger subgraphs remains an open question.

Further interesting lines to follow regard the complexity of approximate counting, counting all approximately densest subgraphs, as well as the complexity of *enumerating* subgraphs.

Our work was strongly motivated also by the practical relevance of counting subgraphs in the dynamic setting. For this reason, we consider an experimental evaluation of dynamic subgraph counting algorithms a relevant and very interesting task for future work.

References

- 1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. If the current clique algorithms are optimal, so is valiant’s parser. *SIAM Journal on Computing*, 47(6):2527–2555, 2018.
- 2 N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, March 1997. doi:10.1007/BF02523189.
- 3 Ziv Bar-Yossef, Ravi Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA ’02*, pages 623–632, USA, 2002. Society for Industrial and Applied Mathematics.
- 4 Thiago Bergamaschi, Monika Henzinger, Maximilian Probst Gutenberg, Virginia Vassilevska Williams, and Nicole Wein. New techniques and fine-grained hardness for dynamic near-additive spanners. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 1836–1855. SIAM, 2021. doi:10.1137/1.9781611976465.110.
- 5 Karl Bringmann, Nick Fischer, and Marvin Künnemann. A fine-grained analogue of schaefer’s theorem in p: Dichotomy of $\exists k\forall$ -quantified first-order graph properties. In *34th Computational Complexity Conference*, pages 1–27. Schloss Dagstuhl, 2019.
- 6 Luciana S. Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. Counting triangles in data streams. In *Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS ’06*, pages 253–262, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1142351.1142388.
- 7 Laxman Dhulipala, Quanquan C. Liu, Julian Shun, and Shangdi Yu. Parallel batch-dynamic k-clique counting. *CoRR*, abs/2003.13585, 2020. arXiv:2003.13585.
- 8 Zdeněk Dvořák and Vojtěch Tůma. A dynamic data structure for counting subgraphs in sparse graphs. In Frank Dehne, Roberto Solis-Oba, and Jörg-Rüdiger Sack, editors, *Algorithms and Data Structures*, pages 304–315, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- 9 David Eppstein, Michael T. Goodrich, Darren Strash, and Lowell Trott. Extended dynamic subgraph statistics using h-index parameterized data structures. *Theor. Comput. Sci.*, 447:44–52, 2012. doi:10.1016/j.tcs.2011.11.034.
- 10 David Eppstein and Emma S. Spiro. The h-index of a graph and its application to dynamic subgraph statistics. *J. Graph Algorithms Appl.*, 16(2):543–567, 2012. doi:10.7155/jgaa.00273.
- 11 Kathrin Hanauer, Monika Henzinger, and Qi Cheng Hua. Fully dynamic four-vertex subgraph counting. *CoRR*, abs/2106.15524, 2021. arXiv:2106.15524.
- 12 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*, pages 21–30, 2015.
- 13 Monika Henzinger, Andrea Lincoln, and Barna Saha. The complexity of average-case dynamic subgraph counting. In *Proceedings of the Thirty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2022, Alexandria, Virginia, USA, January 9-12, 2022*. SIAM, 2022. to appear.
- 14 Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Counting Triangles under Updates in Worst-Case Optimal Time. In Pablo Barcelo and Marco Calautti, editors, *22nd International Conference on Database Theory (ICDT 2019)*, volume 127 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:18, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ICDT.2019.4.
- 15 Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Maintaining triangle queries under updates. *ACM Trans. Database Syst.*, 45(3):11:1–11:46, 2020. doi:10.1145/3396375.

- 16 T. Kloks, D. Kratsch, and H. Müller. Finding and counting small induced subgraphs efficiently. In Manfred Nagl, editor, *Graph-Theoretic Concepts in Computer Science*, pages 14–23, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.
- 17 William L Kocay. Some new methods in reconstruction theory. In *Combinatorial Mathematics IX*, pages 89–114. Springer, 1982.
- 18 Tamara G. Kolda, Ali Pinar, Todd Plantenga, C. Seshadhri, and Christine Task. Counting triangles in massive graphs with mapreduce. *SIAM Journal on Scientific Computing*, 36(5):S48–S77, 2014. doi:10.1137/13090729X.
- 19 F. Le Gall. Powers of tensors and fast matrix multiplication. In K. Nabeshima, K. Nagasaka, F. Winkler, and Á. Szántó, editors, *International Symposium on Symbolic and Algebraic Computation, ISSAC '14, Kobe, Japan, July 23-25, 2014*, pages 296–303. ACM, 2014. doi:10.1145/2608628.2608664.
- 20 Andrea Lincoln, Virginia Vassilevska Williams, and Ryan Williams. Tight hardness for shortest cycles and paths in sparse graphs. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1236–1252. SIAM, 2018.
- 21 Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *VLDB J.*, 29(2-3):595–618, 2020. doi:10.1007/s00778-019-00548-x.

Temporal Unit Interval Independent Sets

Danny Hermelin 

Department of Industrial Engineering and Management,
Ben-Gurion University of the Negev, Beer-Sheva, Israel

Yuval Itzhaki 

Faculty IV, Algorithmics and Computational Complexity, TU Berlin, Germany

Hendrik Molter 

Department of Industrial Engineering and Management,
Ben-Gurion University of the Negev, Beer-Sheva, Israel

Rolf Niedermeier 

Faculty IV, Algorithmics and Computational Complexity, TU Berlin, Germany

Abstract

Temporal graphs have been recently introduced to model changes to a given network that occur throughout a fixed period of time. We introduce and investigate the `TEMPORAL Δ INDEPENDENT SET` problem, a temporal variant of the well known `INDEPENDENT SET` problem. This problem is *e.g.* motivated in the context of finding conflict-free schedules for maximum subsets of tasks, that have certain (changing) constraints on each day they need to be performed. We are specifically interested in the case where each task needs to be performed in a certain time-interval on each day and two tasks are in conflict on a day if their time-intervals overlap on that day. This leads us to considering `TEMPORAL Δ INDEPENDENT SET` on the restricted class of temporal unit interval graphs, *i.e.*, temporal graphs where each layer is unit interval.

We present several hardness results for this problem, as well as two algorithms: The first is a constant-factor approximation algorithm for instances where τ , the total number of time steps (layers) of the temporal graph, and Δ , a parameter that allows us to model some tolerance in the conflicts, are constants. For the second result we use the notion of order preservation for temporal unit interval graphs that, informally, requires the intervals of every layer to obey a common ordering. We provide an FPT algorithm parameterized by the size of minimum vertex deletion set to order preservation.

2012 ACM Subject Classification Theory of computation \rightarrow Graph algorithms analysis; Theory of computation \rightarrow Fixed parameter tractability; Mathematics of computing \rightarrow Discrete mathematics

Keywords and phrases Temporal Graphs, Vertex Orderings, Order Preservation, Interval Graphs, Algorithms and Complexity

Digital Object Identifier 10.4230/LIPIcs.SAND.2022.19

Funding D. Hermelin and H. Molter are supported by the ISF, grant No. 1070/20.

Acknowledgements The authors want to thank anonymous SAND reviewers for their constructive comments.

1 Introduction

Suppose there are n postdocs, each requesting access to your lab in the next τ days for conducting their experiments. Each postdoc submitted at the beginning of the semester an application form which specifies a τ -day schedule for lab experiments, where in each day, the postdoc's schedule specifies a single uninterrupted time-interval for their experiment. All of the n postdocs are very promising, and you wish to make sure that at least k of them are able to conduct their research throughout the entire time period of τ days. However, since the lab is small, it is preferable that no two postdocs share it at the same time. How can you find out if the lab can accept the application of at least k postdocs?



© Danny Hermelin, Yuval Itzhaki, Hendrik Molter, and Rolf Niedermeier;
licensed under Creative Commons License CC-BY 4.0

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).

Editors: James Aspnes and Othon Michail; Article No. 19; pp. 19:1–19:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

19:2 Temporal Unit Interval Independent Sets

This problem can be classically modeled as an INDEPENDENT SET problem on an undirected graph G with n vertices, one for each postdoc, by which two vertices are connected by an edge if at *any* day the lab sessions of the two corresponding postdocs overlap. Given such a graph G with a vertex set V and an edge set E , a solution to INDEPENDENT SET will comprise a subset of vertices $V' \subseteq V$ such that no two vertices in V' are connected by an edge in E . In case that there exists an independent set V' with a cardinality at least k , there exist k postdocs whose research projects can be scheduled over the next τ days such that there are no conflicting lab sessions on any day.

However, by considering the static graph G comprising all conflicts on its own, we lose all the daily information of each postdoc. This can be a serious hindrance if we are willing to allow some leeway in the way we schedule the lab sessions. For example, given that practically every postdoc in the group is known to skip their lab session every once in a while, one might want to allow some overlaps in the schedule. Thus, one could assume that if the experiments of two postdocs do not overlap in more than Δ consecutive days, then they can still be scheduled together. This leads us to the TEMPORAL Δ INDEPENDENT SET problem which we introduce below.

If we wish to retain the daily information of each postdoc, we naturally have to generalize our graph to a *temporal* graph. Temporal graphs generalize static graphs by adding a discrete temporal dimension to the edge set. Formally, a temporal graph $\mathcal{G} = (V, \mathcal{E}, \tau)$ is an ordered triple consisting of a set V of vertices, a set $\mathcal{E} \subseteq \binom{V}{2} \times \{1, 2, \dots, \tau\}$ of *time-edges*, and a maximal time label $\tau \in \mathbb{N}$. A temporal graph can be regarded as a set of τ consecutive *time steps*, in which each step is a static graph. For $t \in \{1, \dots, \tau\}$, we define the t -th layer as $G_t = (V, E_t)$, where $E_t = \{\{u, v\} : (\{u, v\}, t) \in \mathcal{E}\}$. We refer to Casteigts *et al.* [6], Flocchini *et al.* [13], Kostakos [29], Latapy *et al.* [30] and Michail [34] for a more detailed background on temporal graphs.

We next extend the notion of independent sets to temporal graphs. We say a vertex set V' is a Δ -*independent set* in a temporal graph $\mathcal{G} = (V, \mathcal{E}, \tau)$ if V' is an independent set in the edge-*intersection* graph of every Δ consecutive time steps of \mathcal{G} . That is, for any pair of distinct vertices $u \neq v \in V'$ and $t \in \{1, \dots, \tau - \Delta + 1\}$, there exists a $t' \in \{t, \dots, t + \Delta - 1\}$ such that $\{u, v\} \notin E_{t'}$. We call this edge-intersection graph of every Δ consecutive time steps $G = (V, \bigcup_{i=1}^{\tau-\Delta+1} \bigcap_{j=i}^{i+\Delta-1} E_j)$ the *conflict graph*. With this notion in mind, we can now introduce the main problem we deal with in this paper. We give an example in Figure 1.

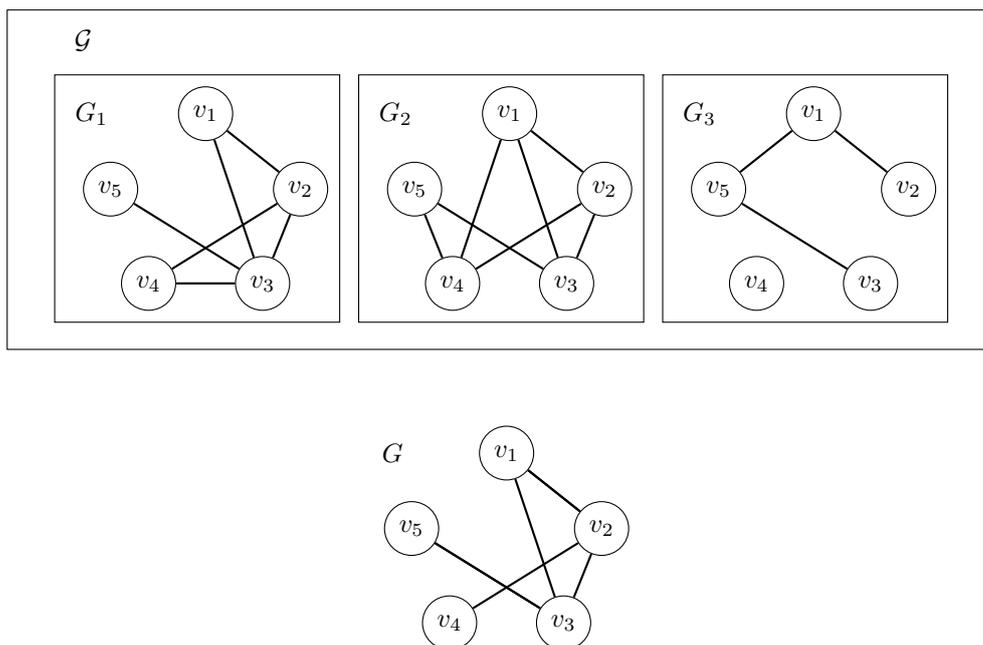
TEMPORAL Δ INDEPENDENT SET

Input: A temporal graph $\mathcal{G} = (V, \mathcal{E}, \tau)$ and two integers $k, \Delta \in \mathbb{N}$.

Question: Is there set $V' \subseteq V$ of vertices such that $|V'| \geq k$ and V' is an independent set in

$$\text{the conflict graph } G = (V, \bigcup_{i=1}^{\tau-\Delta+1} \bigcap_{j=i}^{i+\Delta-1} E_j)?$$

Temporal interval graphs. Recall our initial problem of allocating lab space to the n postdocs. Observe that in this setting, each daily conflict graph G_t (corresponding to layer t of the input temporal graph) can also be represented by a set of n intervals, where each interval indicates a lab session's time-interval of the corresponding scientist on day t . Such a representation is called an *interval representation* of G_t , and it is a unique property of *interval graphs*. As first defined by Hajós [21], a graph belongs to the class of interval graphs if there exists a mapping of its vertices to a set of intervals over a line such that two vertices are adjacent if and only if their corresponding intervals overlap. Interval graphs are used to model many natural phenomena which occur along the line of a one-dimensional axis, and have various applications in scheduling [3], computational biology [26], and more.



■ **Figure 1** An example of TEMPORAL Δ INDEPENDENT SET instance with a temporal graph \mathcal{G} with three layers and $\Delta = 2$. The graph G is the conflict graph of the instance. The vertex subset $V' = \{v_1, v_4, v_5\}$ is a maximum sized solution for this instance.

An important subclass of interval graphs is the class of *unit interval graphs*: A graph G is a *unit interval graph* if it has interval representation where all intervals are of the same length. It is well-known that this graph class is equivalent to the class of *proper interval graphs*, graphs with interval representation where no interval is properly contained in another [36]. The restriction to unit interval graphs is quite natural for our lab allocation problem, since in many cases one can assume that all experiments take roughly the same time. Thus, we will also focus on the TEMPORAL Δ INDEPENDENT SET problem restricted to temporal unit interval graphs.

We focus in this paper on temporal interval graphs, that is temporal graphs \mathcal{G} where each layer is an interval graph. More specifically, most of our work is focused on temporal unit interval graphs, *i.e.*, the case where each layer is a unit interval graph. It should be clear at this point that assuming that all lab sessions take the same amount of time, our lab allocation problem is precisely the TEMPORAL Δ INDEPENDENT SET problem restricted to temporal unit interval graphs.

Geometric interpretation. The restriction of TEMPORAL Δ INDEPENDENT SET to *temporal interval graphs* gives rise to an elegant and useful geometric interpretation. Let us first consider the case of $\Delta = 1$. A *t-track (unit) interval* is a union of t (unit) intervals, one each from t parallel lines, and a *family of t-track (unit) intervals* is a set of t -track (unit) intervals which have intervals all from the same t parallel lines. Clearly, the set of all τ intervals corresponding to a single vertex in G can be represented by a τ -track interval, and so for $\Delta = 1$, the conflict graph G of \mathcal{G} is an intersection graph of a family of τ -track unit intervals. For $\Delta > 1$, we need to use hyperrectangles and hypercubes instead of intervals and unit intervals. We define *t-track hyperrectangles (hypercubes)* and *families of t-track hyperrectangles (hypercubes)* in the natural manner. In this way, we get that our conflict graph G is an intersection graph of a family of $(\tau - \Delta + 1)$ -track Δ -dimensional hypercubes.

Order-preserving temporal interval graphs. Considering our introductory example, it may be a reasonable assumption that some postdocs generally prefer to conduct their experiments in the morning while others prefer to work in the evenings. In this scenario, we have a natural ordering on the time-intervals of the postdocs that stays the same or at least does not change much over the time period of τ days. We use the notion of *order-preserving temporal graphs* to formalize this setting. Order preservation on temporal interval graphs was first introduced by Fluschnik *et al.* [14]. A *temporal interval graph* is said to be *order-preserving* if it admits a vertex ordering $<_V$ such that each of its time steps can be represented by an interval model such that *both* the right-endpoints and left-endpoints are ordered by $<_V$. Fluschnik *et al.* [14] show that the recognition of order-preserving temporal unit interval graphs can be done in linear time, and also offer a metric to measure the distance of a temporal interval graph from being order-preserving, which they call the “shuffle number”. It measures the maximum pairwise disagreements in the vertex ordering of any two consecutive layers. We propose an alternative metric to measure the distance to order preservation. Our distance is simply the minimum number k of vertices to be deleted in order to obtain an order-preserving temporal interval graph, and we call it the order-preserving vertex deletion (OPVD) metric.

1.1 Our results

We present both hardness results and positive results regarding TEMPORAL Δ INDEPENDENT SET in temporal unit interval graphs. Our initial hardness results in Section 2.2 are based on adaptations of the hardness result of Marx [31] for INDEPENDENT SET on axis parallel squares in the plane, and Jiang [25] for INDEPENDENT SET on t -track graphs. These hardness results apply for quite restricted instances of temporal interval graphs. For instance, we show that our problem is NP-hard even for $\tau = 2$.

In Section 3, we present an approximation algorithm for MAXIMUM TEMPORAL Δ INDEPENDENT SET (the canonical optimization variant of TEMPORAL Δ INDEPENDENT SET where we want to maximize the independent set size) restricted to temporal unit interval graphs. The algorithm exploits the geometric interpretation of the problem discussed above to achieve an approximation factor of $(\tau - \Delta + 1) \cdot 2^\Delta$ in polynomial time, and can also be extended to the weighted case.

In Section 4, we turn to discuss order-preserving temporal unit interval graphs. As mentioned above, this concept was introduced by Fluschnik *et al.* [14] for temporal interval graphs. We show that computing the OPVD set (*i.e.*, the set of vertices whose removal leaves an order-preserving graph) of a unit interval temporal graph is NP-hard. We complement this result by providing an FPT algorithm for computing an OPVD set when parameterized by the solution size. This leads to an FPT algorithm for Δ INDEPENDENT SET in temporal unit interval graph when parameterized by minimum OPVD set.

1.2 Related Work

By now, there is already a significant body of research related to temporal graphs in general [6, 13, 29, 34], as well as graph problems cast onto the temporal setting [2, 4, 14, 23, 39, 33, 32]. To the best of our knowledge, the problem of TEMPORAL Δ INDEPENDENT SET has not been studied previously, but our definition is highly inspired by the TEMPORAL Δ CLIQUE problem [4, 23, 39].

The classical static INDEPENDENT SET problem is clearly a special case of TEMPORAL Δ INDEPENDENT SET when $\tau = 1$. While INDEPENDENT SET is NP-complete for general undirected graphs [15], it is solvable in linear time on interval graphs and some of their

generalizations [16, 24, 37]. Thus, TEMPORAL Δ INDEPENDENT SET on temporal interval graph is linear time solvable when $\tau = 1$. Moreover, for larger values of τ , the TEMPORAL 1 INDEPENDENT SET problem is a special case of INDEPENDENT SET in τ -interval graphs [19]. Bar-Yehuda *et al.* [3] presented a 2τ approximation algorithm for INDEPENDENT SET in τ -interval graphs, while Fellows *et al.* [12] and Jiang [25] studied this problem from the perspective of parameterized complexity.

When $\tau = \Delta$, TEMPORAL Δ INDEPENDENT SET is a special case of INDEPENDENT SET on intersection graphs of τ -dimensional hyperrectangles. Marx [31] showed that INDEPENDENT SET is NP-complete and W[1]-hard with respect to the solution size when restricted to the intersection graphs of axis-parallel unit squares in the plane. Chlebík and Chlebíková [8] proved, for instance, that MAXIMUM INDEPENDENT SET is APX-hard for intersection graphs of d -dimensional *rectangles*, yet on such graphs the optimal solution can be approximated within a factor of d [1]. On intersection graphs of d -dimensional *squares* MAXIMUM INDEPENDENT SET admits a *polynomial time approximation scheme (PTAS)* for a constant d [7, 22, 28].

2 Preliminaries and Basic Results

In this section, we first introduce all temporal graph notation used in this work and then present some basic hardness results for our problem. We use standard notation and terminology from parameterized complexity theory [10].

2.1 Notation and Definitions

Let $a, b \in \mathbb{N}$ such that $a < b$. We use the notation $[a : b]$ as shorthand for $\{x \mid x \in \mathbb{N} \wedge a \leq x \leq b\}$. We denote $[a : b]$ by $[b]$ when $a = 1$. For $a, b \in \mathbb{R}$ such that $a < b$ we denote with $[a, b] \subseteq \mathbb{R}$ the set of real numbers $\{x \mid x \in \mathbb{R} \wedge a \leq x \leq b\}$.

Let $G = (V, E)$ denote an undirected graph, where V denotes the set of vertices and $E \subseteq \{\{v, w\} \mid v, w \in V, v \neq w\}$ denotes the set of edges. For a graph G , we also write $V(G)$ and $E(G)$ to denote the set of vertices and the set of edges of G , respectively. We denote $n := |V|$. Given an ordering $<_V$ over the vertices V of a graph $G = (V, E)$ in which v_i is the i -th vertex in the ordering, we denote by $V_{[a:b]}$ the set $\{v_i \mid i \in [a : b]\}$ and by $G_{[a:b]}$ the graph induced by $V_{[a:b]}$. We use the notation $\text{index}_{<_V}(v)$ to return the ordinal position of v in $<_V$.

An undirected *temporal graph* $\mathcal{G} = (V, \mathcal{E}, \tau)$ is an ordered triple consisting of a set V of vertices, a set $\mathcal{E} \subseteq \binom{V}{2} \times [\tau]$ of *time-edges*, and a maximal time label $\tau \in \mathbb{N}$. Given a temporal graph $\mathcal{G} = (V, \mathcal{E}, \tau)$, we denote by E_t the set of all edges that are available at time t , that is, $E_t := \{\{v, w\} \mid (\{v, w\}, t) \in \mathcal{E}\}$ and by G_t the t -th *layer* of \mathcal{G} , that is, $G_t := (V, E_t)$. We also denote by

- $G_1 \cap G_2$ the *edge-intersection graph* of G_1 and G_2 , formally $G_1 \cap G_2 := (V, E_1 \cap E_2)$,
- $G_1 \cup G_2$ the *edge-union graph* of G_1 and G_2 , formally $G_1 \cup G_2 := (V, E_1 \cup E_2)$, and
- $\mathcal{G} - V'$ the *temporal graph* induced by $V \setminus V'$, formally $\mathcal{G} - V' := (V \setminus V', \mathcal{E}', \tau)$ with $\mathcal{E}' = \{\{v, u\}, t \mid v, u \in V \setminus V' \wedge (\{v, u\}, t) \in \mathcal{E}\}$.

Interval graphs. An undirected graph is an *interval graph* if there exists a mapping from its vertices to intervals on the real line so that two vertices are adjacent if and only if their intervals intersect [17]. Such a representation is called an *intersection model* or an *interval representation*. Formally, given a graph $G = (V, E)$, an *interval representation* is a mapping $\rho : V \rightarrow I \subseteq \mathbb{R}$ of each vertex $v \in V$ to an interval $\rho(v)$ such that $E = \{\{v, u\} \mid v, u \in V \wedge \rho(v) \cap \rho(u) \neq \emptyset\}$. We denote by $\text{right}_\rho(u)$ and by $\text{left}_\rho(u)$ the real value of the right and

left endpoints of v 's associated interval on the interval representation ρ ; the subscript ρ will be omitted if it is clear from the context to which representation we refer. We denote by $\rho(G)$ the entire representation of G .

Unit interval graphs. An interval graph is a *unit interval graph* if it has an interval representation in which all intervals have exactly the same length. It is exactly the class of interval graphs which have a proper interval representation, a representation in which no interval is properly contained within another [36].

2.2 Basic Hardness Results

Since TEMPORAL Δ INDEPENDENT SET generalizes the classic INDEPENDENT SET problem, it is clearly NP-hard [27], W[1]-hard when parameterized by the solution size k [10], and does not admit any efficient approximation algorithms as well. In this section, we show that TEMPORAL Δ INDEPENDENT SET remains computationally hard when restricted to temporal unit interval graphs with a constant number of layers.

We begin with the case that $\Delta = 1$. Here the conflict graph is simply the union of all layers of \mathcal{G} . Thus, as mentioned in Section 1, the class of all possible conflict graphs is precisely the class of τ -track unit intervals. Thus, the following hardness result directly follows from the known hardness results for INDEPENDENT SET in 2-track unit interval graph [3, 12, 25].

► **Proposition 1.** *TEMPORAL Δ INDEPENDENT SET on temporal unit interval graphs is NP-hard, APX-hard, and W[1]-hard with respect to the solution size k for $\tau \geq 2$ and $\Delta = 1$.*

Now that we have seen hardness for $\tau \geq 2$ and $\Delta = 1$, we proceed with case of $\Delta = \tau$, by which the class of all conflict graphs is precisely the class of τ -dimensional hypercubes (intersection) graphs. Marx [31] showed that INDEPENDENT SET on axis-parallel unit squares in the plane is NP-hard and W[1]-hard with respect to the solution size. For our setting, this result implies the following.

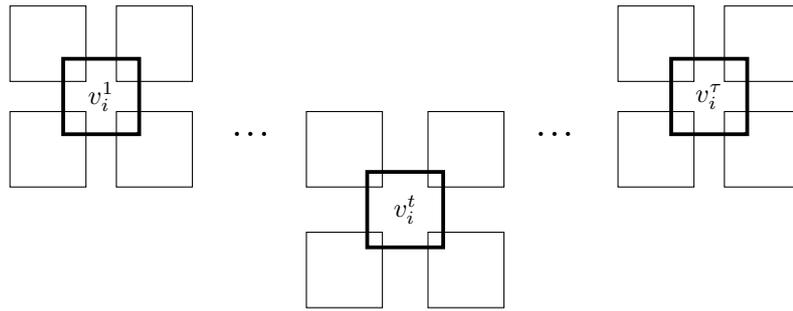
► **Proposition 2.** *TEMPORAL Δ INDEPENDENT SET on temporal unit interval graphs is NP-hard and W[1]-hard with respect to the solution size k for $\tau = \Delta \geq 2$.*

Note however, that the problem admits a PTAS in this case, whenever $\tau = \Delta = \mathcal{O}(1)$ [11].

3 Approximation Algorithms

In this section, we present our polynomial-time approximation algorithm for WEIGHTED MAXIMUM TEMPORAL Δ INDEPENDENT SET on temporal unit interval graphs. We will use the geometric representation of our conflict graph discussed in Section 1. Recall that the conflict graph G has a $(\tau - \Delta + 1)$ -track Δ -dimensional hypercube representation. This geometric property of the conflict graph can be exploited in the development of an approximation algorithm, since we can upper-bound the size of the largest independent set in the neighborhood of any vertex of G .

► **Lemma 3.** *Let G be the conflict graph of MAXIMUM TEMPORAL Δ INDEPENDENT SET on a temporal unit interval graph. For each vertex it holds that the largest independent set in the graph induced by $N[v]$ has at most $2^\Delta(\tau - \Delta + 1)$ vertices.*



■ **Figure 2** A set of τ axis parallel Δ -hypercubes cannot be intersected by more than $2^\Delta \tau$ disjoint axis parallel Δ hypercubes.

Proof. Let $\rho(G)$ denote the $\tau - \Delta + 1$ -track Δ -dimensional hypercube family representation of G . Furthermore, for a vertex v of G , let $\rho(v) \in \rho(G)$ denote the $\tau - \Delta + 1$ -track Δ -dimensional hypercube corresponding to v in $\rho(G)$. Consider a single hypercube C of $\rho(v)$. Observe that any other hypercube in $\rho(G)$ intersecting C must include a corner of C . Consequently, there are at most 2^Δ disjoint hypercubes in $\rho(G)$ intersecting C , and so at most $2^\Delta(\tau - \Delta + 1)$ disjoint $\tau - \Delta + 1$ -track hypercubes intersect $\rho(v)$ (see Figure 2). The lemma thus follows. ◀

Lemma 3 allows to adopt a greedy approach for solving WEIGHTED MAXIMUM TEMPORAL Δ INDEPENDENT SET on temporal unit interval graphs. We iteratively put the largest-weight vertex into the independent set and remove its neighborhood afterwards. This gives rise to Algorithm 1 below.

■ **Algorithm 1** Maximum Neighborhood Weight Algorithm.

Input: An undirected graph $G = (V, E)$
Output: An independent set
while V is not empty **do**
 Pick vertex v with maximum $w(v)$ and add it to S .
 Remove $N[v]$ from V .
end while
return S

We show that the greedy strategy described in Algorithm 1 terminates within $\mathcal{O}(n^2)$ time, and achieves an approximation factor of $(\tau - \Delta + 1) \cdot 2^\Delta$.

► **Theorem 4.** *WEIGHTED MAXIMUM TEMPORAL Δ INDEPENDENT SET can be approximated within a factor of $(\tau - \Delta + 1) \cdot 2^\Delta$ in $\mathcal{O}(n^2)$ time on temporal unit interval graphs.*

Proof. Let \mathcal{G} be the input temporal unit interval graph of WEIGHTED MAXIMUM TEMPORAL Δ INDEPENDENT SET. We can compute the conflict graph G in linear time. We know that G is a $(\tau - \Delta + 1)$ -track Δ -dimensional hypercube intersection graph. Given $G = (V, E)$ with weight function w as an input, the output of Algorithm 1 will be obviously an independent set. From Lemma 3 we know that it cannot have smaller weight than $2^\Delta(\tau - \Delta + 1)$ times of the optimal solution’s weight. This is due to the fact that at each step of Lemma 3 we add one vertex to the solution and remove its neighbors from the candidate set. The optimal solution is of course an independent set, hence at each step we remove from the candidate set no more than $2^\Delta(\tau - \Delta + 1)$ vertices which are members of the optimal solution. Since we

pick in each step a maximum weight vertex, assuming the worst case in which we throw out a $2^\Delta(\tau - \Delta + 1)$ -sized set of vertices with equal weight, we cannot have our solution's weight smaller than this ratio. Suppose that Algorithm 1 terminated after p steps, this means that there are p vertices in the solution and no more than $2^\Delta p(\tau - \Delta + 1)$ vertices in the optimal solution. \blacktriangleleft

4 Order-Preserving Temporal Interval Graphs

In this section, we investigate the computational complexity of TEMPORAL Δ INDEPENDENT SET on so-called order-preserving temporal unit interval graphs. The concept of order preservation was introduced by Fluschnik *et al.* [14]. In Section 4.1, we show that TEMPORAL Δ INDEPENDENT SET can be solved in polynomial time on order-preserving temporal graphs and along the way we generalize the concept to temporal (non-unit) interval graphs. In Section 4.2, we show how to solve TEMPORAL Δ INDEPENDENT SET on non-order-preserving temporal interval graphs via a “distance-to-triviality” parameterization [18]. To this end, we also give an FPT algorithm to compute a minimum vertex deletion set to order preservation with the set size as a parameter. Finally in Section 4.3, we show that computing a minimum vertex deletion set to order preservation is NP-hard.

4.1 A Polynomial-Time Algorithm for Order-Preserving Temporal Interval Graphs

We say an interval graph *agrees on* or is *compatible with* a total order if it has an interval intersection model where the right endpoints of the intervals agree with the total order. Formally, an interval graph G agrees on $<_V$ if there exists an interval representation ρ for G such that for every two vertices $\forall v, u \in V$ whose ranking fulfills $v <_V u$, it holds that $\text{right}_\rho(v) < \text{right}_\rho(u)$. We call such ordering *right-endpoints (RE) orderings*. Clearly, any right-endpoints ordering is also a *left-endpoints* ordering of the mirrored intersection model.

► **Definition 5.** *A temporal interval graph is order-preserving if all of its layers agree on a single RE ordering.*

Order-preserving temporal unit interval graphs can be recognized in linear time and a corresponding vertex ordering can be computed in linear time as well [14]. The computational complexity of recognizing order-preserving temporal *interval* graphs remains open.

In the following, we show that RE orderings are preserved under both intersection and union of interval graphs. This means that the conflict graph of an RE order-preserving temporal graph is an interval graph that as well agrees on the RE ordering. We demonstrate this claim for interval graphs in Lemmas 6 and 7. We start with showing that the intersection of two interval graphs that agree on an RE ordering is again an interval graph that agrees on the ordering.

► **Lemma 6.** *Let G_1 and G_2 be interval graphs that agree on the total ordering $<_V$. Then $G_1 \cap G_2$ is an interval graph that agrees on $<_V$.*

Proof. Given two interval graphs which agree on an RE ordering, we can normalize their representations such that for each vertex, the right endpoints in both representations are the same. To compute an intersection model for their intersection graph, we can map each vertex to the intersection of their intervals in both representations. We then show that this mapping is an interval representation of the edge intersection graph.

Let V be a vertex set of size n and let $<_V$ be a total ordering on V such that for all $i, j \in [n]$ it holds $j < i \Leftrightarrow v_j <_V v_i$. Since both G_1 and G_2 agree on $<_V$, we can normalize their interval representations so that the right endpoint of the interval associated with each vertex lies on a natural number between 1 and n according to its ordinal position in $<_V$, formally $\text{index}_{<_V}(v_i) = i$. Alternatively, we can say that for an interval graph G_t an interval representation ρ_t exists such that each $v \in V$ is mapped to an interval of the form $\rho_t(v) = [a_v, \text{index}_{<_V}(v)]$ with $a_v \in \mathbb{R}$. In this normalized representation, the left endpoint of the interval lies on the real line between two natural numbers and is by definition smaller than the right endpoint, that is, $a_v < \text{index}_{<_V}(v)$.

Let ρ be a mapping from the vertex set V to a set of points on \mathbb{R} such that it holds $\rho(v) = \rho_1(v) \cap \rho_2(v)$. To show that ρ is an interval representation of $G_1 \cap G_2$ we first show that $\rho(v)$ is a continuous interval for any $v \in V$, and that for any two vertices $v, u \in V$ it holds that $\rho(v) \cap \rho(u) \neq \emptyset \Leftrightarrow \rho_1(v) \cap \rho_1(u) \neq \emptyset \wedge \rho_2(v) \cap \rho_2(u) \neq \emptyset$.

By definition $\rho_1(v)$ and $\rho_2(v)$ are both intervals on the real line, they are therefore convex sets. As the mapping $\rho(v)$ is an intersection of two convex sets, it must as well be a convex set and therefore it is interval on the real line.

Let $v_j <_V v_i$, we show that if $\rho(v_i) \cap \rho(v_j) \neq \emptyset$ then both $\rho_1(v_i) \cap \rho_1(v_j) \neq \emptyset$ and $\rho_2(v_i) \cap \rho_2(v_j) \neq \emptyset$ must hold. As the interval representations ρ_1 and ρ_2 are both normalized, it immediately follows that $\text{right}_{\rho_1}(v_i) = \text{right}_{\rho_2}(v_i) = i$. Since both are closed intervals we know that either $\rho_1(v) \subseteq \rho_2(v)$ or $\rho_2(v) \subseteq \rho_1(v)$. Without loss of generality, let $\rho_1(v) \subseteq \rho_2(v)$; it follows that $\rho(v) = \rho_1(v)$. This means that if $\rho_2(v_i) \cap \rho_2(v_j) = \emptyset$, then also $\rho_1(v_i) \cap \rho_1(v_j) = \emptyset$ and therefore $\rho(v_i) \cap \rho(v_j) = \emptyset$. Regardless, it must hold that $j \in \rho_1(v_j) \cap \rho_2(v_j)$ as both interval representations of v_j have j as the right endpoint; it follows that $j \in \rho(v_j)$. This shows that if $j \in \rho(v_i)$ then $j \in \rho_1(v_i)$ and $j \in \rho_2(v_i)$. Therefore, for any $v_j <_V v_i$, if $\rho(v_i) \cap \rho(v_j) \neq \emptyset$, then both $\rho_1(v_i) \cap \rho_1(v_j) \neq \emptyset$ and $\rho_2(v_i) \cap \rho_2(v_j) \neq \emptyset$.

Suppose that $\rho(v_i) \cap \rho(v_j) = \emptyset$, but both $\rho_1(v_i) \cap \rho_1(v_j) \neq \emptyset$ or $\rho_2(v_i) \cap \rho_2(v_j) \neq \emptyset$. This contradicts that $v_j <_V v_i$ because if $\rho_1(v_i)$ contains any point $a \in \rho_1(v_j)$ with $a < j$, then it must contain also j because $\rho_1(v_i)$ is convex.

We have therefore an interval representation ρ which represents the graph $G_1 \cap G_2$ because $\rho(v) \cap \rho(u) \Leftrightarrow \{v, u\} \in E_1 \wedge \{v, u\} \in E_2$ for any $v, u \in V$. Notice that $G_1 \cap G_2$ agrees on $<_V$ because $\text{right}_{\rho}(v_i) = i$. ◀

Next, we show that the union of two interval graphs agreeing on an RE ordering yields an interval graph that also agrees on the ordering.

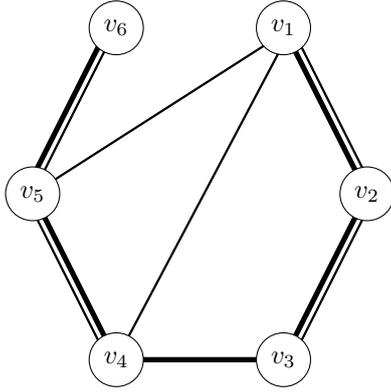
► **Lemma 7.** *Let G_1 and G_2 be interval graphs that agree on the total ordering $<_V$. The union $G = G_1 \cup G_2$ is an interval graph that agrees on $<_V$.*

Proof. The main concept of the proof is analogous to the one for Lemma 6. Given two interval graphs which agree on an RE ordering, we can normalize their representations such that for each vertex, the right endpoints in both representations are the same. To compute an intersection model for their *union* graph, we can map each vertex to the *union* of their intervals in both representations. We then show that this mapping is an interval representation of the edge-*union* graph.

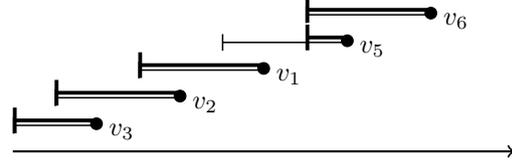
Let ρ be a mapping from the vertex set V to a set of points on \mathbb{R} such that it holds $\rho(v) = \rho_1(v) \cup \rho_2(v)$. To show that ρ is an interval representation of $G_1 \cup G_2$ we first show that $\rho(v)$ is a continuous interval for any $v \in V$, and that for any two vertices $v, u \in V$ it holds that $\rho(v) \cap \rho(u) \neq \emptyset \Leftrightarrow \rho_1(v) \cap \rho_1(u) \neq \emptyset \vee \rho_2(v) \cap \rho_2(u) \neq \emptyset$.

By definition $\rho_1(v)$ and $\rho_2(v)$ are both closed and normalized intervals on the real line such that $\text{right}_{\rho_1}(v_i) = \text{right}_{\rho_2}(v_i) = i$. As we observed in Lemma 6, since both intervals have the same right endpoint it holds that either $\rho_1(v) \subseteq \rho_2(v)$ or $\rho_2(v) \subseteq \rho_1(v)$. Without loss of generality, assume that $\rho_2(v) \subseteq \rho_1(v)$, it follows that $\rho(v) = \rho_1(v) = \rho_1(v) \cup \rho_2(v)$.

19:10 Temporal Unit Interval Independent Sets



(a) Graph representations of the two interval graphs



(b) Two intersection models of $G_1 - \{v_4\}$ and $G_2 - \{v_4\}$ which are compatible with $\prec_{V'}$

■ **Figure 3** Two interval graphs, G_1 (thick) and G_2 (thin), that *do not* have a common RE ordering. The vertex subset $\{v_4\}$ is an OPVD set of the temporal graph $\mathcal{G} = [G_1, G_2]$ as both $G_1 - \{v_4\}$ and $G_2 - \{v_4\}$ agree on $\prec_{V'} = [v_3, v_2, v_1, v_5, v_6]$. Two compatible interval representations are illustrated in (b).

Suppose that $\rho(v_i) \cap \rho(v_j) \neq \emptyset$, but both $\rho_1(v_i) \cap \rho_1(v_j) = \emptyset$ or $\rho_2(v_i) \cap \rho_2(v_j) = \emptyset$. This contradicts that $v_j \prec_V v_i$ because if $\rho(v_i)$ contains any point $a \in \rho(v_j)$ with $a < j$, then it must contain also j because $\rho(v_i)$ is convex. If $j \in \rho(v_i)$ then trivially $j \in \rho_1(v_i)$.

If $\rho(v_i) \cap \rho(v_j) = \emptyset$ but either $\rho_1(v_i) \cap \rho_1(v_j) \neq \emptyset$ or $\rho_2(v_i) \cap \rho_2(v_j) \neq \emptyset$, then it contradicts the fact that either $\rho_1(v_i) \subseteq \rho_2(v_i) = \rho(v_i)$ or that $\rho_2(v_i) \subseteq \rho_1(v_i) = \rho(v_i)$.

We have therefore an interval representation ρ which represents the graph $G_1 \cup G_2$ because $\rho(v) \cap \rho(u) \Leftrightarrow \{v, u\} \in E_1 \vee \{v, u\} \in E_2$ for any $v, u \in V$. Notice that $G_1 \cup G_2$ agrees on \prec_V because $\text{right}_\rho(v_i) = i$. ◀

Using both Lemmas 6 and 7 we arrive at the following corollary.

► **Corollary 8.** *Let \mathcal{G} be an order-preserving temporal interval graph of a TEMPORAL Δ INDEPENDENT SET instance. Then the conflict graph of \mathcal{G} is an interval graph.*

Since INDEPENDENT SET can be solved in linear time on interval graphs [16, 37], we immediately arrive at our main result of this subsection.

► **Theorem 9.** *TEMPORAL Δ INDEPENDENT SET on order-preserving temporal interval graphs is solvable in linear time.*

4.2 An FPT-Algorithm for Vertex Deletion to Order Preservation

Now we generalize Theorem 9 and show how to solve TEMPORAL Δ INDEPENDENT SET on *almost* order-preserving temporal unit interval graphs, that is, graphs that most of their vertices agree on a common ordering. To this end, we define a *distance* of a temporal graph to order preservation. This distance is measured by the size of the minimum vertex set that obstructs the compatibility of a total RE order of a temporal interval graph. We define it as follows and give an illustration in Figure 3.

► **Definition 10 (OPVD).** *Let $\mathcal{G} = (V, \mathcal{E}, \tau)$ be a temporal interval graph. A vertex deletion set for order preservation (OPVD) is a set of vertices $V' \subseteq V$ such that $\mathcal{G} - V'$ is order-preserving.*

The size of the minimum OPVD set measures how many vertices obstruct a total RE order for a temporal interval graph. We denote the cardinality of the minimum OPVD by ℓ . A brute-force algorithm checks every subset of the vertex set to find a solution to TEMPORAL Δ INDEPENDENT SET. Given an ℓ -sized OPVD set we can brute-force the power set of the OPVD (which has size 2^ℓ) and then check against the rest of the order-preserving graph in polynomial time.

► **Theorem 11.** *TEMPORAL Δ INDEPENDENT SET can be decided in $2^\ell \cdot n^{\mathcal{O}(1)}$ time when given a size- ℓ OPVD set of the input temporal graph.*

Proof. The idea is as follows. Given an order-preserving vertex deletion set S of size ℓ , we brute-force its power set. Let G_{op} be the conflict graph of $\mathcal{G} - S$. The graph G_{op} is, by definition, an interval graph. For each subset X of S we compute G_X as $G_{\text{op}} - N_{G_{\text{op}}}(X)$, the neighbors of X from the conflict interval graph. As G_X is an interval graph, we can compute a maximum independent set V' of G_X in linear time, then check in quadratic time whether $X \cup V'$ is an independent set of size k in the conflict graph of \mathcal{G} . If $X \cup V'$ is an independent set of size at least k , then we have a yes-instance.

Any independent set of size k must clearly be divisible into two subsets, a subset of $X \subseteq S$ (that includes the trivial subset) and a subset of $V \setminus S$. Any independent set on \mathcal{G} must be also an independent set on the subgraph induced by $V \setminus S$. If we exhaust all of the subsets of S and do not find an independent set of size at least $k - |X|$ on $\mathcal{G} - (S \cup N_{G_{\text{op}}}[X])$ for $X \subseteq S$, then we can conclude that such set does not exist. In such case the instance is a no-instance. The power set of S is of size 2^ℓ , which means it takes $2^\ell \cdot n^{\mathcal{O}(1)}$ time to exhaust all subsets of S . ◀

Since the FPT algorithm for TEMPORAL Δ INDEPENDENT SET parameterized by the minimum OPVD ℓ behind Theorem 11 requires access to an ℓ -sized OPVD set, we present an FPT-algorithm to compute a minimum OPVD for a given temporal unit interval graph. We do this by providing a reduction to the so-called CONSECUTIVE ONES SUBMATRIX BY COLUMN DELETIONS problem, for which efficient algorithms are known [9, 35].

Before we describe the reduction, we give an alternative characterization of order-preserving temporal unit interval graphs. We will use this characterization in our FPT-algorithm to compute a minimum OPVD. As we show in the next lemma, a temporal unit interval graph \mathcal{G} is order-preserving if and only if its vertices vs. maximal cliques matrix has the so-called *consecutive ones property*¹ (C1P). Note that it is known that the vertices vs. neighborhoods matrix also has the consecutive ones property in this case [14].

► **Lemma 12.** *A temporal unit interval graph is order-preserving if and only if its vertices vs. maximal cliques matrix has the consecutive ones property.*

Proof. Testing for the consecutive ones property for a matrix can be done in linear time [38]. To test a temporal unit interval graph for order preservation we compute its vertices vs. maximal cliques matrix and test it for the consecutive ones property. We say that \mathcal{G} 's set of maximal cliques \mathcal{C} is the union of sets of maximal cliques of each layer of \mathcal{G} . The vertices vs. maximal cliques matrix M is a binary matrix in which $M_{i,j} = 1$ if and only if the vertex $v_i \in V$ is a member of $C_j \in \mathcal{C}$. It is left to show that a temporal unit interval graph \mathcal{G} is order-preserving if and only if its vertices vs. maximal cliques matrix has the consecutive ones property.

¹ A 0-1-matrix has the *consecutive ones property* if there exists a permutation of the columns such that in each row all ones appear consecutively.

19:12 Temporal Unit Interval Independent Sets

(\Rightarrow) If \mathcal{G} is order-preserving, then there exists an ordering $<_V$ such that every layer has an interval representation ρ in which the right endpoints of all intervals agree on $<_V$. Let M 's columns be ordered by $<_V$. If M is not in its petrie form ², then it must mean that there exists a clique C in \mathcal{C} whose members are not consecutive in $<_V$. In other words, there exist $u, v, w \in V$ such that $u <_V w <_V v$, for which $u, v \in C$ and $w \notin C$. Since u and v are adjacent and $u <_V v$, we know that $\text{left}(v) < \text{right}(u)$. We know also that the length of $\rho(v)$ is exactly 1. This definitely means that v and w intersect because $\text{right}(w) \in [\text{right}(u), \text{right}(v)]$. However since $w \notin C$, w and u cannot be adjacent. This is a contradiction since $\text{right}(v) - \text{right}(u) < 1$ and $\text{right}(w) - \text{left}(w) = 1$. If \mathcal{G} is order-preserving, then M must have the consecutive ones property.

(\Leftarrow) If M has the consecutive ones property, then there exists an ordering $<_V$ so that the vertices vs. maximal cliques matrix M_t of every layer $G_t \in \mathcal{G}$ is in its petrie form, when its columns are permuted according to $<_V$. Let $\mathcal{J}_t(v)$ be the union of all maximal cliques in layer G_t which contain v . We know that for every $v \in V$ the vertices of $\mathcal{J}_t(v)$ are consecutive in $<_V$ [5]. Let $\text{index}(v)$ be the index of v in $<_V$ and let $\rho_{G_t}(v) = [\min\{\text{index}(u) \mid u \in \mathcal{J}_t(v)\} - 1 + \text{index}(v) \cdot \varepsilon, \text{index}(v)]$, for some $0 < \varepsilon < 1/|V|$. First, note that no two intervals are contained in each other. This means that there is an equivalent interval representation where all intervals have unit length [36].

By showing $\{u, v\} \in E_t \Leftrightarrow \rho_t(v) \cap \rho_t(u) \neq \emptyset$ we effectively show that ρ_{G_t} is an interval representation of G_t . If $\{u, v\} \in E_t$, then there must exist a maximal clique C so that $u, v \in C$ and thus $u \in \mathcal{J}_t(v)$ and $v \in \mathcal{J}_t(u)$. Assume $u <_V v$, then $\text{left}_{\rho_{G_t}}(v) \leq \text{right}_{\rho_{G_t}}(u)$ and by that $\rho_{G_t}(u) \cap \rho_{G_t}(v) \neq \emptyset$. If $\{u, v\} \notin E_t$ then there is no clique C so that $u, v \in C$. Assume that $u <_V v$, then $\rho_{G_t}(u) \cap \rho_{G_t}(v) \neq \emptyset$ if and only if $\text{left}_{\rho_{G_t}}(v) \leq \text{right}_{\rho_{G_t}}(u)$. This cannot be because $\text{left}_{\rho_{G_t}}(v)$ is exactly the right endpoint of v 's lowest neighbors in $<_V$. If $\text{right}_{\rho_{G_t}}(u)$ is right of v 's lowest neighbor's right endpoint, then the vertices of $\mathcal{J}_t(v)$ are not consecutive in $<_V$. This contradicts the fact that M has the consecutive ones property. \blacktriangleleft

Since there is a bijection between the columns of the vertices vs. maximal cliques matrix M of \mathcal{G} and \mathcal{G} 's vertices, we can use M as input for the CONSECUTIVE ONES SUBMATRIX BY COLUMN DELETIONS problem and apply existing algorithms for that problem [9, 35] to obtain a vertex-maximal temporal subgraph of \mathcal{G} that is order-preserving. This allows us to obtain the following result.

► Theorem 13. *A minimum OPVD for a given temporal unit interval graph can be computed in $10^\ell n^{\mathcal{O}(1)}$ time, where ℓ is the size of a minimum OPVD.*

Proof. In Lemma 12 we have shown that a temporal unit interval graph is order-preserving if and only if its vertices vs. maximal cliques matrix has the consecutive ones property. We provide a reduction to the CONSECUTIVE ONES SUBMATRIX BY COLUMN DELETIONS problem. Formally, in CONSECUTIVE ONES SUBMATRIX BY COLUMN DELETIONS we are given a binary matrix $M \in \{0, 1\}^{m \times n}$ and are asked whether there exists a submatrix M' with the consecutive ones property, such that M' is obtained with not more than ℓ column deletions from M . CONSECUTIVE ONES SUBMATRIX BY COLUMN DELETIONS is known to be FPT with respect to the column deletion set size and it can be decided in $10^\ell n^{\mathcal{O}(1)}$ time [9, 35].

Let $\mathcal{G} = (V, \mathcal{E})$ be a temporal unit interval graph with \mathcal{C} as maximal cliques set. Let M be the vertices vs. maximal cliques matrix of \mathcal{G} . If M does not have the consecutive ones property, then we can find a set of ℓ columns in $10^\ell n^{\mathcal{O}(1)}$ time so that when deleted from M ,

² A 0-1-matrix is in its *petrie form* (if it has one) if the columns are permuted in a way such that the ones appear consecutively in all rows.

the resulting matrix M' has the consecutive ones property. The columns of M are mapped to vertices of V , the image of the deleted columns V' is the OPVD set. We can find in linear time an ordering \langle'_V of M' columns such that M' is in its petrie form. All layers of the graph $\mathcal{G} - V'$ agree on \langle'_V . ◀

This provides us with an efficient algorithm for TEMPORAL Δ INDEPENDENT SET on “almost” ordered temporal unit interval graph. Namely, find in $10^\ell n^{\mathcal{O}(1)}$ time a minimum OPVD set in the input temporal unit interval graph using Theorem 13, then decide in $2^\ell n^{\mathcal{O}(1)}$ time if we have a yes-instance of TEMPORAL Δ INDEPENDENT SET using Theorem 11. Overall, we arrive at the following result.

► **Corollary 14.** *TEMPORAL Δ INDEPENDENT SET can be decided in $10^\ell \cdot n^{\mathcal{O}(1)}$ time if the input temporal graph is a temporal unit interval graph, where ℓ is the size of a minimum OPVD of the input temporal graph.*

4.3 NP-Hardness of Vertex Deletion to Order Preservation

Finally, we show that computing a minimum OPVD for a given temporal unit interval graph is NP-hard. This complements Theorem 13 as it implies that we presumably cannot improve Theorem 13 to a polynomial-time algorithm.

► **Theorem 15.** *Computing a minimum OPVD for a given temporal unit interval graph is NP-hard.*

Proof. To show NP-hardness, we present a polynomial time many-one reduction from the NP-complete CONSECUTIVE ONES SUBMATRIX BY COLUMN DELETIONS problem [20] to the problem of computing an OPVD of size at most ℓ for a given temporal unit interval graph. Note that this implies NP-hardness of the optimization problem of finding a minimum OPVD.

Formally, in CONSECUTIVE ONES SUBMATRIX BY COLUMN DELETIONS we are given a binary matrix $M \in \{0, 1\}^{m \times n}$ and are asked whether there exists a submatrix M' with the consecutive ones property, such that M' is obtained with not more than ℓ column deletions from M . Note that we can assume w.l.o.g. that there are at least two ones in each row of M , otherwise we can delete the row since its ones are consecutive for all permutations of the columns.

Our reduction works as follows. Given a binary matrix $M \in \{0, 1\}^{m \times n}$ with m rows and n columns, we create a temporal graph \mathcal{G} with n vertices $V = \{1, \dots, n\}$, one for each column, and m layers, one for each row. In each layer G_t for $1 \leq t \leq m$, we add an edge between vertices i and j if $M_{t,i} = 1$ and $M_{t,j} = 1$. This finished the construction of \mathcal{G} , which can clearly be done in polynomial time.

Next, we argue that \mathcal{G} is a temporal unit interval graph. To this end, note that every layer G_t of \mathcal{G} is a single clique (consisting of vertices i with $M_{t,i} = 1$) and some isolated vertices (the vertices i with $M_{t,i} = 0$). Hence, we can clearly find a unit interval representation for every layer G_t of \mathcal{G} .

To prove the correctness of the reduction, we first observe that M is the vertices vs. maximal cliques matrix of \mathcal{G} : there is exactly one non-trivial maximal clique in each layer G_t containing the vertices i with $M_{t,i} = 1$. We show ℓ columns can be deleted from M such that the remaining matrix M' has the consecutive ones property if and only if \mathcal{G} admits an OPVD of size ℓ .

(\Rightarrow) Assume there are ℓ columns that can be deleted from M such that the remaining matrix M' has the consecutive ones property. Then M' corresponds to vertices vs. maximal cliques matrix of \mathcal{G}' which is obtained from \mathcal{G} by removing the ℓ vertices corresponding to the deleted columns of M . By Lemma 12 we have that \mathcal{G}' is an order-preserving temporal unit interval graph. It follows that the removed vertices form an OPVD of size ℓ for \mathcal{G} .

(\Leftarrow) Assume \mathcal{G} admits an OPVD X of size ℓ . Then let M' be the matrix obtained from M by deleting the ℓ columns corresponding to the vertices in X . Now we have that M' is the vertices vs. maximal cliques matrix of $\mathcal{G} - X$, which is an order-preserving temporal unit interval graph. By Lemma 12 we have that M' has the consecutive ones property and hence that (M, ℓ) is a yes-instance of CONSECUTIVE ONES SUBMATRIX BY COLUMN DELETIONS. \blacktriangleleft

5 Conclusion

We introduced a naturally motivated temporal version of the classic INDEPENDENT SET problem, called TEMPORAL Δ INDEPENDENT SET, and investigated its computational complexity. Herein, we focused on the case where all layers of the input temporal graph are unit interval graphs. After establishing computational hardness results, we showed that MAXIMUM TEMPORAL Δ INDEPENDENT SET admits a polynomial-time $(\tau - \Delta + 1) \cdot 2^\Delta$ -approximation. Furthermore, we presented a polynomial-time algorithm for TEMPORAL Δ INDEPENDENT SET when restricted to so-called order-preserving temporal interval graphs and generalized it to an FPT-algorithm for the vertex deletion distance to order preservation. The latter heavily relies on our result that order preservation is retained under edge-union and edge-intersection, which is of independent interest since it may also be useful in the context of related problem such as TEMPORAL Δ CLIQUE [4, 23, 39].

An immediate future work direction is to generalize our results for temporal (non-unit) interval graphs. For most of our results this question remains open. We believe that our approximation algorithm does not easily adapt. In fact even for two layers it is unclear how to approximate MAXIMUM TEMPORAL Δ INDEPENDENT SET. Our FPT-algorithm for TEMPORAL Δ INDEPENDENT SET parameterized by the vertex deletion distance to order preservation generalizes to the non-unit interval case assuming the deletion set is part of the input. We leave for future research how to efficiently compute a minimum vertex deletion set to order preservation for temporal interval graphs.

References

- 1 Karhan Akcoglu, James Aspnes, Bhaskar DasGupta, and Ming-Yang Kao. Opportunity cost algorithms for combinatorial auctions. In *Computational Methods in Decision-Making, Economics and Finance*, pages 455–479. Springer, 2002.
- 2 Eleni C. Akrida, George B. Mertzios, Paul G. Spirakis, and Viktor Zamaraev. Temporal vertex cover with a sliding time window. *Journal of Computer and System Sciences*, 107:108–123, 2020.
- 3 Reuven Bar-Yehuda, Magnús M. Halldórsson, Joseph Naor, Hadas Shachnai, and Irina Shapira. Scheduling split intervals. *SIAM Journal on Computing*, 36(1):1–15, 2006.
- 4 Matthias Bentert, Anne-Sophie Himmel, Hendrik Molter, Marco Morik, Rolf Niedermeier, and René Saitenmacher. Listing all maximal k -plexes in temporal graphs. *Journal of Experimental Algorithmics (JEA)*, 24:1–27, 2019.
- 5 Kellogg S. Booth and George S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *Journal of Computer and System Sciences*, 13(3):335–379, 1976.

- 6 Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012.
- 7 Timothy M. Chan. Polynomial-time approximation schemes for packing and piercing fat objects. *Journal of Algorithms*, 46(2):178–189, 2003.
- 8 Miroslav Chlebík and Janka Chlebíková. Approximation hardness of optimization problems in intersection graphs of d -dimensional boxes. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '05)*, pages 267–276. SIAM, 2005.
- 9 Michael Dom, Jiong Guo, and Rolf Niedermeier. Approximation and fixed-parameter algorithms for consecutive ones submatrix problems. *Journal of Computer and System Sciences*, 76(3-4):204–221, 2010.
- 10 Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Springer, 2013.
- 11 Thomas Erlebach, Klaus Jansen, and Eike Seidel. Polynomial-time approximation schemes for geometric intersection graphs. *SIAM Journal on Computing*, 34(6):1302–1323, 2005.
- 12 Michael R. Fellows, Danny Hermelin, Frances Rosamond, and Stéphane Vialette. On the parameterized complexity of multiple-interval graph problems. *Theoretical Computer Science*, 410(1):53–61, 2009.
- 13 Paola Flocchini, Bernard Mans, and Nicola Santoro. On the exploration of time-varying networks. *Theoretical Computer Science*, 469:53–68, 2013.
- 14 Till Fluschnik, Hendrik Molter, Rolf Niedermeier, Malte Renken, and Philipp Zschoche. Temporal graph classes: A view through temporal separators. *Theoretical Computer Science*, 806:197–218, 2020.
- 15 Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.
- 16 Fănică Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *Journal of Combinatorial Theory, Series B*, 16(1):47–56, 1974.
- 17 Paul C. Gilmore and Alan J. Hoffman. A characterization of comparability graphs and of interval graphs. *Canadian Journal of Mathematics*, 16:539–548, 1964.
- 18 Jiong Guo, Falk Hüffner, and Rolf Niedermeier. A structural view on parameterizing problems: Distance from triviality. In *Proceedings of the 1st International Workshop on Parameterized and Exact Computation (IWPEC '04)*, pages 162–173. Springer, 2004.
- 19 András Gyárfás and Douglas West. Multitrack interval graphs. *Congressus Numerantium* 109, 1995.
- 20 Mohammad Taghi Hajiaghayi and Yashar Ganjali. A note on the consecutive ones submatrix problem. *Information processing letters*, 83(3):163–166, 2002.
- 21 György Hajós. Über eine Art von Graphen. *Internationale Mathematische Nachrichten*, 11(65), 1957.
- 22 Monika Henzinger, Stefan Neumann, and Andreas Wiese. Dynamic Approximate Maximum Independent Set of Intervals, Hypercubes and Hyperrectangles. In *Proceedings of the 36th International Symposium on Computational Geometry (SoCG '20)*, volume 164 of *LIPIcs*, pages 51:1–51:14. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.
- 23 Anne-Sophie Himmel, Hendrik Molter, Rolf Niedermeier, and Manuel Sorge. Adapting the Bron–Kerbosch algorithm for enumerating maximal cliques in temporal graphs. *Social Network Analysis and Mining*, 7(1):35:1–35:16, 2017.
- 24 Wen-Lian Hsu and Jeremy P. Spinrad. Independent sets in circular-arc graphs. *Journal of Algorithms*, 19(2):145–160, 1995.
- 25 Minghui Jiang. On the parameterized complexity of some optimization problems related to multiple-interval graphs. *Theoretical Computer Science*, 411(49):4253–4262, 2010.
- 26 Deborah Joseph, Joao Meidanis, and Prasoon Tiwari. Determining DNA sequence similarity using maximum independent set algorithms for interval graphs. In *Proceedings of the 3rd Scandinavian Workshop on Algorithm Theory (SWAT '92)*, pages 326–337. Springer, 1992.

19:16 Temporal Unit Interval Independent Sets

- 27 Richard M Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Springer, 1972.
- 28 Sanjeev Khanna, Shan Muthukrishnan, and Mike Paterson. On approximating rectangle tiling and packing. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, volume 95, page 384. SIAM, 1998.
- 29 Vassilis Kostakos. Temporal graphs. *Physica A: Statistical Mechanics and its Applications*, 388(6):1007–1023, 2009.
- 30 Matthieu Latapy, Tiphaine Viard, and Clémence Magnien. Stream graphs and link streams for the modeling of interactions over time. *Social Network Analysis and Mining*, 8(1):61:1–61:29, 2018.
- 31 Dániel Marx. Efficient approximation schemes for geometric problems? In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA '05)*, pages 448–459. Springer, 2005.
- 32 George B. Mertzios, Hendrik Molter, Rolf Niedermeier, Viktor Zamaraev, and Philipp Zschoche. Computing maximum matchings in temporal graphs. In *Proceedings of the 37th International Symposium on Theoretical Aspects of Computer Science (STACS '20)*, volume 154 of *LIPICs*, pages 27:1–27:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
- 33 George B. Mertzios, Hendrik Molter, and Viktor Zamaraev. Sliding window temporal graph coloring. *Journal of Computer and System Sciences*, 120:97–115, 2021.
- 34 Othon Michail. An introduction to temporal graphs: An algorithmic perspective. *Internet Mathematics*, 12(4):239–280, 2016.
- 35 N.S. Narayanaswamy and R. Subashini. Obtaining matrices with the consecutive ones property by row deletions. *Algorithmica*, 71(3):758–773, 2015.
- 36 Fred S. Roberts. Indifference graphs. Proof techniques in graph theory. In *Proceedings of the Second Ann Arbor Graph Conference, Academic Press, New York*, 1969.
- 37 Donald J. Rose, Robert Endre Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5(2):266–283, 1976.
- 38 Alan Tucker. A structure theorem for the consecutive 1's property. *Journal of Combinatorial Theory, Series B*, 12(2):153–162, 1972.
- 39 Tiphaine Viard, Matthieu Latapy, and Clémence Magnien. Computing maximal cliques in link streams. *Theoretical Computer Science*, 609:245–252, 2016.

Search by a Metamorphic Robotic System in a Finite 3D Cubic Grid

Ryonosuke Yamada ✉

Graduate School of Information Science and Electrical Engineering,
Kyushu University, Fukuoka, Japan

Yukiko Yamauchi ✉

Faculty of Information Science and Electrical Engineering,
Kyushu University, Fukuoka, Japan

Abstract

We consider search in a finite 3D cubic grid by a *metamorphic robotic system* (MRS), that consists of anonymous modules. A module can perform a sliding and rotation while the whole modules keep connectivity. As the number of modules increases, the variety of actions that the MRS can perform increases. The *search problem* requires the MRS to find a target in a given finite field. Doi et al. (SSS 2018) demonstrate a necessary and sufficient number of modules for search in a finite 2D square grid. We consider search in a finite 3D cubic grid and investigate the effect of common knowledge. We consider three different settings. First, we show that three modules are necessary and sufficient when all modules are equipped with a *common compass*, i.e., they agree on the direction and orientation of the x , y , and z axes. Second, we show that four modules are necessary and sufficient when all modules agree on the direction and orientation of the vertical axis. Finally, we show that five modules are necessary and sufficient when all modules are not equipped with a common compass. Our results show that the shapes of the MRS in the 3D cubic grid have richer structure than those in the 2D square grid.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Computer systems organization → Robotic autonomy

Keywords and phrases Distributed system, metamorphic robotic system, search, and 3D cubic grid

Digital Object Identifier 10.4230/LIPIcs.SAND.2022.20

Funding This work was supported by JSPS KAKENHI Grant Numbers JP18H03202 and JST SICORP Grant Number JPMJSC1806, Japan.

1 Introduction

Swarm intelligence has shed light to collective behavior of autonomous entities with simple rules, such as ant, boid, and particles. The notion is applied to a collection of robots and swarm robotics has attracted much attention in the past two decades. Each autonomous element of the system is called a robot, module, agent, process, and sensor, and a variety of swarm robot systems have been investigated such as the *autonomous mobile robot system* [16], the *population protocol model* [3], the *programmable particles* [5], *Kilobot* [15], and *3D Catoms* [17]. Dumitrescu et al. considered the *metamorphic robotic system* (MRS), that consists of a collection of *modules* in the infinite 2D square grid [10, 9]. The modules are *anonymous*, i.e., they are indistinguishable. They are *autonomous* and *uniform*, i.e., each module autonomously decides its movement by a common algorithm. They are *oblivious*, i.e., each module has no memory of past. Thus, each module decides its behavior by observing other modules in nearby cells. Each module can perform a *sliding* to a side-adjacent cell and a *rotation* by 90 degrees around a cell. The modules must keep *connectivity*, which is defined by side-adjacency of cells occupied by modules. The authors considered *reconfiguration*, that requires the MRS to change the initial shape to a specified final shape [10]. They showed that any horizontally convex connected initial shape of an MRS can be transformed to any



© Ryonosuke Yamada and Yukiko Yamauchi;

licensed under Creative Commons License CC-BY 4.0

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).

Editors: James Aspnes and Othon Michail; Article No. 20; pp. 20:1–20:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

convex final shape via a straight chain shape. Later Dumitrescu and Pach showed that any connected initial shape can be transformed to any connected final shape via a straight chain shape [8]. In other words, the MRS has the ability of “universal reconfiguration.”

Reconfiguration can generate dynamic behavior of the MRS. Dumitrescu et al. demonstrated that the MRS can move forward by repeating a reconfiguration [9]. They showed a reconfiguration that realizes the fastest *locomotion*. Doi et al. pointed out that the oblivious modules can use the shape of the MRS as global memory and the MRS can solve more complicated problems as the number of modules increases. They investigated *search* in a finite 2D square grid, that requires the MRS to find a target cell in a finite rectangle *field* [7]. Each module does not know the position of the target cell or the initial configuration of the MRS. They showed that if the modules agree on the cardinal directions (i.e., north, south, east and west), three modules are necessary and sufficient, otherwise five modules are necessary and sufficient. Nakamura et al. considered *evacuation* of the MRS from a finite rectangular field in the 2D square grid [14]. There is a hole (i.e., two side-adjacent cells) on the wall of the field, and the MRS is required to exit from it from an arbitrary initial position and arbitrary initial shape. They showed that two modules are necessary and sufficient when the modules agree on the cardinal directions, otherwise four modules are necessary and sufficient.

In this paper, we investigate the effect of common knowledge on search by the MRS in the 3D cubic grid. We consider the following three cases: (i) modules equipped with a common *compass* (i.e., they agree on the direction and orientation of x , y , and z axes), (ii) modules equipped with a common vertical axis (i.e., they agree on the direction and orientation of z axis), and (iii) modules not equipped with a common compass (i.e., they have no agreement on directions and orientations). We demonstrate that three modules are necessary and sufficient when the modules are equipped with a common compass and five modules are necessary and sufficient when the modules are not equipped with a common compass. The numbers of sufficient modules in the 3D cubic grid are the same as those in the 2D square grid [7] because the MRS has more states in the 3D cubic grid than in the 2D square grid. For the intermediate case with a common vertical axis, we demonstrate that four modules are necessary and sufficient. Thus, our results in the 3D cubic grid show a smooth trade-off between the computational power of the MRS and common knowledge among modules, that the previous results in the 2D square grid could not find. We present search algorithms for these three settings and show the necessity by examining the state transition graph of the MRS.

Related work. Reconfiguration of swarm robot systems have been discussed for the MRS [8, 10], autonomous mobile robots [11, 16] and programmable particles [6, 12]. Michail et al. considered the *programmable matter system*, that is similar to the MRS and investigated reconfiguration by rotations only and that by rotations and slidings [13]. They showed that the combination of rotations and slidings guarantees universal reconfiguration, while rotations only cannot. They also presented $O(n^2)$ -time reconfiguration algorithm by rotations and slidings, where n is the number of computing entities. Almethen et al. considered reconfiguration by *line-pushing*, where each module is equipped with the ability of pushing a line of modules [2]. They presented $O(n \log n)$ -time universal reconfiguration algorithm that does not promise connectivity of intermediate shapes and $O(n\sqrt{n})$ -time reconfiguration algorithm that transforms a diagonal line into a straight chain with preserving connectivity. The same authors later showed that their programmable matter system has the ability of universal reconfiguration and $O(n\sqrt{n})$ -time reconfiguration algorithm together with $\Omega(n \log n)$ lower bound [1].

Little has been discussed for memory complexity of swarm robot systems. The autonomous mobile robot system consider extreme cases, where all robots are equipped with either no memory or unlimited memory [16]. Das et al. introduced the *luminous robot model*, where each robot is equipped with a light [4]. A luminous robot can change the color of its light and the color of the light can be observed by other robots. Thus, a luminous robot has a state. The authors showed that the robots can be synchronized by a constant number of colors. Doi. et al. pointed out that the number of memory-less modules of the MRS can be considered as an indicator of memory complexity [7]. The existing two papers [7, 14] demonstrated the relationship between search and evacuation in 2D square grid. Finally, the programmable particle system [5] consider computing entities with constant-size memory.

2 Preliminary

We consider a *metamorphic robotic system* (MRS) in a finite 3D cubic grid. A metamorphic robotic system consists of a collection of anonymous (i.e., indistinguishable) *modules*. A module can observe the positions of other modules in nearby cells, computes its next movement with a common algorithm, and performs the movement.

Each cell of the 3D cubic grid can adopt at most one module at a time. Cell (x, y, z) is the cell surrounded by grid points (x, y, z) , $(x + 1, y, z)$, $(x, y + 1, z)$, $(x, y, z + 1)$, $(x + 1, y + 1, z)$, $(x + 1, y, z + 1)$, $(x, y + 1, z + 1)$, and $(x + 1, y + 1, z + 1)$. Cells $(x + 1, y, z)$, $(x, y + 1, z)$, $(x, y, z + 1)$, $(x - 1, y, z)$, $(x, y - 1, z)$, $(x, y, z - 1)$ are *side-adjacent* to cell (x, y, z) . We consider the positive x direction as East, the positive y direction as North, and the positive z direction as Up.

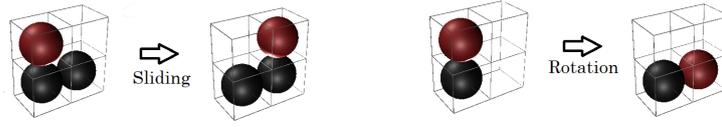
The MRS moves in a finite *field*, which is a cuboid of width w , depth d and height h with its two diagonal cells being $(0, 0, 0)$ and $(w - 1, d - 1, h - 1)$. We consider two types of *planes*; the first type is a set of cells forming a plane perpendicular to one of the x , y , and z axes. The second type is a set of cells parallel to one of the x , y , and z axes and diagonal to the remaining two axes. For example, $\{(x, y, z) | y = s\}$ for some $s \in \mathcal{Z}$ is a plane of the first type and $\{(x, y, z) | x + y = s'\}$ for some $s' \in \mathcal{Z}$ is a plane of the second type. A *line* of cells is a set of cells forming a horizontal or vertical line on a plane. For example, $\{(x, y, z) | y = u, z = v\}$ for some $u, v \in \mathcal{Z}$ is a line and $\{(x, y, z) | x + y = u', z = v'\}$ for some $u', v' \in \mathcal{Z}$ is a line.

The field is surrounded by six planes, which we call *walls*. More precisely, the walls are $\{(x, y, z) | x = -1\}$ (the West wall), $\{(x, y, z) | y = -1\}$ (the South wall), $\{(x, y, z) | z = -1\}$ (the Bottom wall), $\{(x, y, z) | x = w\}$ (the East wall), $\{(x, y, z) | y = d\}$ (the North wall), and $\{(x, y, z) | z = h\}$ (the Top wall).

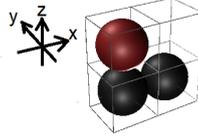
All modules synchronously perform observation, computation, and movement in each discrete time $t = 0, 1, 2, \dots$. A *configuration* of the MRS is the set of cells occupied by the modules. We say two modules are *side-adjacent* if they are in the two side-adjacent cells. We also say that a module m is side-adjacent to cell c if the cell occupied by m is side-adjacent to c . Given a configuration of the MRS, consider a graph where each vertex corresponds to a module and there is an edge between two vertices if the corresponding modules are side-adjacent. If this graph is connected, we say the MRS is *connected*.

A module can perform two types of movements, *sliding* and *rotation*.

1. Sliding: When two modules m_i and m_j are side-adjacent, another module m_k can move from a cell side-adjacent to m_i to an empty cell side-adjacent to m_k and m_j along m_i and m_j . During the movement, m_i and m_j cannot perform any movement. See Figure 1 as an example.



■ **Figure 1** Sliding and rotation. The red modules perform movements.



■ **Figure 2** Example of an observation at the red module with a local coordinate system.

2. Rotation: When two modules m_i and m_j are side-adjacent, m_i can move to a cell side-adjacent to m_j by rotating $\pi/2$ in some direction. There are six cells side-adjacent to m_j and m_i can move to four of them by rotation. During the movement, m_j cannot move and the cells that m_i passes must be empty. See Figure 1 as an example.

Note that several modules can move at the same time as long as their moving tracks do not overlap. The modules must keep two types of connectivity at each time step.

1. At the beginning of each time step, the modules must be connected.
2. At each time step, the modules that do not move must be connected.

We assume that each module obtains the result of an observation and moves to the next cell in its *local x - y - z coordinate system*. We assume that the origin of the local coordinate system of a module is its current cell and all local coordinate systems are right-handed. In this paper, we consider three types of MRSs with different degree of agreement on the coordinate system. When all modules agree on the directions and orientations of x , y , and z axes, we say the MRS is equipped with a *common compass*. When all modules do not agree on the directions or orientations of x , y , and z axes, we say the MRS is not equipped with a common compass. Hence, local coordinate systems are not consistent among the modules. As an intermediate model, we consider modules that agree on the direction and orientation of the vertical axis. In this case, we say the MRS is equipped with a *common vertical axis*. The *state* of the MRS is its local shape. If the modules are equipped with a common compass or a common vertical axis, the state of the MRS contains common directions and orientations. Otherwise the state of the MRS does not contain any directions and orientations.

The *search problem* requires the MRS to find a *target* placed at one cell in the field from a given initial configuration. We call the cell containing the target the *target cell*. The MRS *finds* a target when one of its modules enters the target cell.

When a module executes the common algorithm, the input is the *observation* of cells in a cube of size $(2k+1) \times (2k+1) \times (2k+1)$ centered at the module (i.e., its k -neighborhood). The value of k is fixed by the algorithm. A module detects whether each cell in its k -neighborhood is a wall cell or not and whether it is occupied by a module or not. Let C_m be the set of cells occupied by some modules, C_w be the set of wall cells, and C_e be the set of the remaining (i.e., empty) cells of an observation. More precisely, each set of cells is a set of coordinates of the corresponding cells observed in the local coordinate system of the module. For example, in Figure 2, the result of an observation at the red module is $C_m = \{(0, -1, 0), (1, -1, 0)\}$, $C_w = \emptyset$, and C_e is the remaining cells. When a common algorithm outputs coordinate (a, b, c) at a module, the module moves to (a, b, c) in its local coordinate system.

When we describe an algorithm, the elements of C_m , C_w , and C_e are specified in a “canonical coordinate system,” i.e., the global coordinate system. When the modules are equipped with a common compass, without loss of generality, we assume that the common compass is identical to the global coordinate system. Thus, each module computes its movement by checking C_m , C_w , and C_e . When the modules agree on a common vertical axis, without loss of generality, we assume that the vertical axis is identical to z axis of the global coordinate system. Each module computes its movement by rotating the current observation by $\pi/2$, π , $3\pi/2$, and 2π around the common vertical axis and comparing the results with C_m , C_w , and C_e . It selects an output with a movement and if there are multiple outputs with movement it nondeterministically selects one of them. When the modules are not equipped with a common compass, a module checks 24 rotations of the current observation and selects an output in the same way as the above case.

3 Search algorithms for an MRS in a finite 3D cubic grid

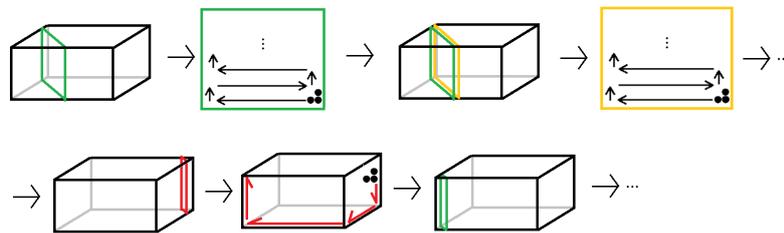
In this section, we present search algorithms with small number of modules. Our proposed algorithms are based on a common strategy. Since the MRS does not know the position of the target cell, we make the MRS visit all cells of the field. The proposed algorithms slice the field into planes and the MRS visits the cells of each plane by sweeping each row or column of the plane. Thus, the proposed algorithms are extensions of the search algorithms by Doi et al. in a finite 2D square grid [7].

3.1 Search with a common compass

We show the following theorem by a search algorithm for the MRS of three modules equipped with a common compass.

► **Theorem 1.** *The MRS of three modules equipped with a common compass can solve the search problem in a finite 3D cubic grid from any initial configuration.*

The proposed algorithm considers planes that is parallel to the z -axis and the angles between it and each of x -axis and y -axis are $\pi/2$. Each plane is represented as $\{(x, y, z) | x + y = s\}$ for $s = 1, 2, \dots$. The MRS moves along each line parallel to the x - y plane $\{(x, y, z) | x + y = s, z = t\}$ for $t = 0, 1, 2, \dots$. Figure 3 shows a moving track of the proposed algorithm.



■ **Figure 3** Search by three modules equipped with a common compass.

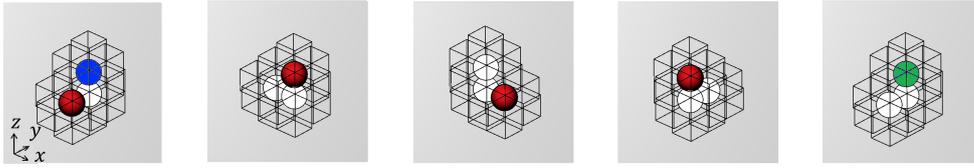
The MRS continues to search each plane until it reaches the northeasternmost plane. Then, it moves along the edges of the field so that it returns to the southwesternmost plane. It starts searching each plane again to visit all cells of the field.

The MRS moves forward or turns by repeating a sequence of movements, that we call a *move sequence*. The proposed algorithm consists of the following move sequences.

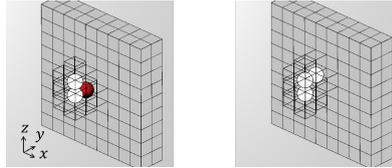
- Move sequence M_{NW} (Figure 4). The blue module is in cell (x, y, z) at first. By this move sequence, the green module reaches cell $(x - 1, y + 1, z)$. By repeating M_{NW} n times, some modules visit the cells $(x - k, y + k, z)$ ($0 \leq k \leq n$). That is, it visits all the cells of the horizontal line $\{(x, y, z) | x + y = s, z = t\}$.
- Move sequence M_{TurnNW} (Figure 5). By this move sequence, the MRS changes its move sequence from M_{NW} to M_{SE} .
- Move sequence M_{SE} (Figure 6). The blue module is in cell (x, y, z) at first. By this move sequence, the green module reaches cell $(x + 1, y - 1, z)$. By repeating M_{SE} n times, some modules visit the cells $(x + k, y - k, z)$ ($0 \leq k \leq n$). That is, it visits all the cells of the horizontal line $\{(x, y, z) | x + y = s, z = t\}$.
- Move sequence M_{TurnSE} (Figure 7). By this move sequence, the MRS changes its move sequence from M_{SE} to M_{NW} .
- Move sequence M_T (Figure 8). By this move sequence, the MRS changes its move sequence from M_{SE} to M_D .
- Move sequence M_D (Figure 9). The blue module is in cell (x, y, z) at first. By this move sequence, the green module reaches cell $(x, y, z - 1)$. By repeating M_D n times, some modules visit the cells $(x, y, z - k)$ ($0 \leq k \leq n$). That is, it visits all the cells of the line $\{(x, y, z) | x = s, y = t\}$.
- Move sequence M_B (Figure 10). By this move sequence, the MRS changes its move sequence from M_D to M_{NW} .
- Move sequence $M_{NECorner}$ (Figure 11). By this move sequence, the MRS changes its move sequence from M_D to $M_{WallBottom}$.
- Move sequence $M_{WallBottom}$ (Figure 12). The blue module is in cell $(x, y, 0)$ at first. By this move sequence, the green module reaches cell $(x, y - 1, 0)$ along the edge. By repeating $M_{WallBottom}$ n times, some modules visit the cells $(x, y - k, 0)$ ($0 \leq k \leq n$). That is, it visits all the cells of the line $\{(x, y, z) | x = s, z = 0\}$.
- Move sequence $M_{SWCorner}$ (Figure 13). By this move sequence, the MRS changes its move sequence from $M_{WallBottom}$ to M_{Up} .
- Move sequence M_{Up} . (Figure 14) The blue module is in cell $(0, 0, z)$ at first. By this move sequence, the green module reaches cell $(0, 0, z + 1)$. By repeating M_{Up} n times, some modules visit the cells $(0, 0, k)$ ($0 \leq k \leq n$). That is, it visits all the cells of the line $\{(x, y, z) | x = 0, y = 0\}$.

The proposed algorithm consists of the following seven steps.

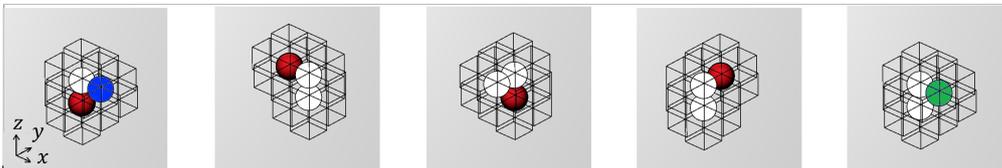
- Step 1** The MRS repeats M_{NW} , that makes it move in the northwest direction along a horizontal line on a plane $\{(x, y, z) | x + y = s\}$ for some s .
- Step 2** When the MRS reaches the north or west wall, it changes the moving direction to southeast by M_{TurnNW} .
- Step 3** The MRS repeats M_{SE} , that makes it move in the southeast direction along a horizontal line on $\{(x, y, z) | x + y = s\}$. This movement makes the MRS move along the same horizontal line as Step 1.
- Step 4** If the MRS is adjacent to the top wall it moves to the plane $\{(x, y, z) | x + y = s + 1\}$. Then, it repeats M_D shown in Figure 9, that makes it move down along the south wall or east wall until it reaches the bottom wall. Then, it leaves the wall by M_B . It starts searching the new plane by repeating Steps 1, 2, 3, and 4. Otherwise, it proceeds to Step 5.
- Step 5** When the MRS reaches the south or east wall, it moves to the row above by M_{TurnSE} . Then, it repeats Steps 1, 2, and 3 so that it visits all cells on the new horizontal line.



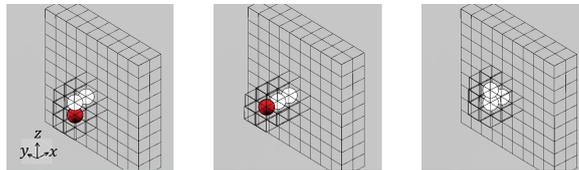
■ **Figure 4** Move to northwest. In each figure, the red module moves. When the blue module is in cell (x, y, z) , after this move sequence, the green module reaches cell $(x - 1, y + 1, z)$.



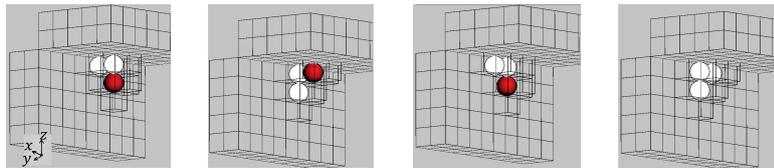
■ **Figure 5** Turn on the north or west wall. In the first figure, the red module moves.



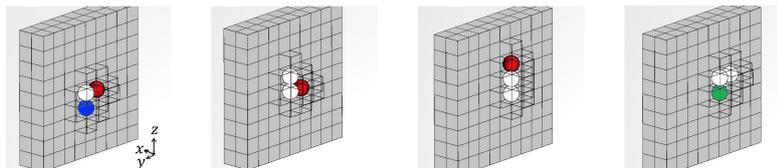
■ **Figure 6** Move to southeast. In each figure, the red module moves. When the blue module is in cell (x, y, z) , after this move sequence, the green module reaches $(x + 1, y - 1, z)$.



■ **Figure 7** Turn on the south or east wall. In each figure, the red module moves.

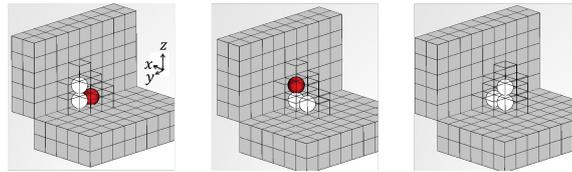


■ **Figure 8** Move around the top of the north or east wall. In each figure, the red module moves.

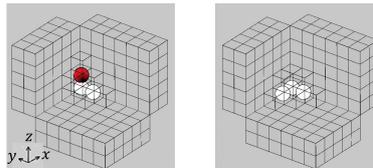


■ **Figure 9** Move down on the north or east wall. In each figure, the red module moves. When the blue module is in cell (x, y, z) , after this move sequence, the green module reaches cell $(x, y, z - 1)$.

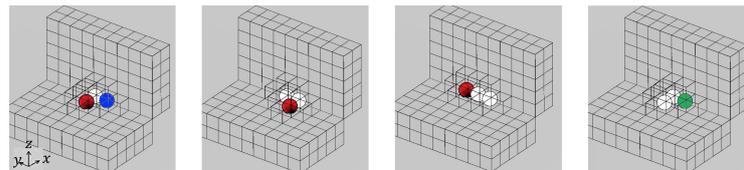
20:8 Search by a Metamorphic Robotic System in a Finite 3D Cubic Grid



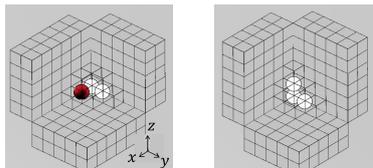
■ **Figure 10** Leaving the bottom of the north or east wall. In each figure, the red module moves.



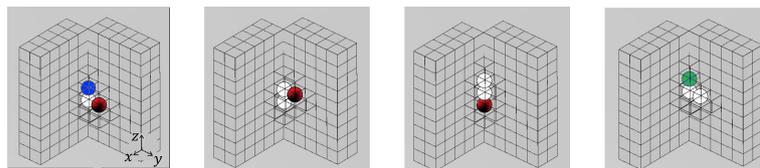
■ **Figure 11** Move on the northeast corner. In the first figure, the red module moves.



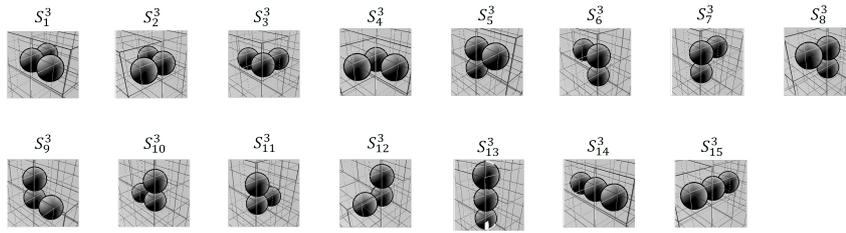
■ **Figure 12** Move along the bottom of the wall. In each figure, the red module moves. When the blue module is in cell $(x, y, 0)$, after this move sequence, the green module reaches cell $(x, y - 1, 0)$.



■ **Figure 13** Move on the southwest corner. In the first figure, the red module moves.



■ **Figure 14** Move up on the southwest corner. In each figure, the red module moves. When the blue module is in cell $(0, 0, z)$, after this move sequence, the green module reaches cell $(0, 0, z + 1)$.



■ **Figure 15** States of the MRS of three modules equipped with a common compass.

Step 6 When the MRS reaches the northeast corner of the top wall, the algorithm sends the MRS back to the southwest corner, where the MRS starts searching by repeating Steps 1 to 5. It moves along the northeast edge until it reaches the northeast corner of the bottom wall by M_D . Then, it moves along the east edge of the bottom wall until it reaches the south east corner of the bottom wall by $M_{NECorner}$ and repeating $M_{WallBottom}$. It moves along the south edge of the bottom wall until it reaches the southwest corner of the bottom wall by repeating $M_{WallBottom}$. Finally, it moves along the southeast edge until it reaches the southwest corner of the top wall by $M_{SWCorner}$ and repeating M_{Up} . Then, the MRS returns to Step 4.

Table 1 and 2 show the input and the output of the proposed algorithm. Each element specifies a part of the input (especially, C_w and C_e), and the MRS does not care whether other cells than those specified are walls or not.

We briefly address the correctness of the proposed algorithm. When the MRS is on a plane $\{(x, y, z) | x + y = s\}$ for some s , it visits all cells in the horizontal line $\{(x, y, z) | x + y = s, z = t\}$ for some t by repeating Steps 1, 2, and 3. Then, it proceeds to the horizontal line $\{(x, y, z) | x + y = s, z = t + 1\}$ by Step 5. By repeating Steps 1, 2, 3, and 5, it eventually reaches the top wall. Then, it starts searching for cells in $\{(x, y, z) | x + y = s + 1\}$ by Step 4 and 6.

Repeating the above movement, the MRS eventually reaches the northeast corner of the top wall. At this point, it may have not yet visited the cells near the south west corner. Steps 6 enables the MRS visit these cells by moving it to the southwest corner of the top wall and starting Step 1 again.

There exist initial configurations that satisfies no condition of Table 1 and 2. We add exceptional transformation rules from such initial configurations. Figure 15 shows all states of three modules equipped with a common compass. Observe that any configuration can be transformed to another one in one time step. (Note that more than one module can move in one time.) Hence, even if the initial state of the MRS does not match any entry of Table 1 and 2, the MRS can be transformed into one of the entries and the MRS can start search from any initial configuration.

3.2 Search with a common vertical axis

We show the following theorem by a search algorithm for the MRS of four modules equipped with a common vertical axis.

► **Theorem 2.** *The MRS of four modules with a common vertical axis can solve a search problem in a finite 3D grid if no pair of modules have an identical observation in an initial configuration.*

■ **Table 1** Search algorithm for the MRS equipped with a common compass (Former part).

	C_m	C_w	C_e	Output
M_{SE}	(0, 0, 1), (1, 0, 1)		(2, 0, 0)	(1, 0, 0)
	(1, 0, 0), (1, 0, -1)			(1, -1, 0)
	(0, 0, 1), (0, -1, 1)		(0, -2, 0)	(0, -1, 0)
	(0, -1, 0), (0, -1, -1)			(1, -1, 0)
M_{TurnSE}	(0, 0, 1), (1, 0, 1)	(2, 0, 0)	(0, -1, 0)	(-1, 0, 1)
	(1, 0, 0), (2, 0, 0)		(0, 0, -1), (0, 0, 1)	(1, 0, 1)
	(0, 0, 1), (0, -1, 1)	(0, -2, 0)		(0, 1, 1)
	(0, -1, 0), (0, -2, 0)	(0, 0, -1)		(0, -1, 1)
M_{NW}	(-1, 0, 0), (-1, 0, 1)		(0, -1, 0)	(-1, 1, 0)
	(0, 0, -1), (0, 1, -1)		(0, 2, 0)	(0, 1, 0)
	(0, 1, 0), (0, 1, 1)		(1, 0, 0)	(-1, 1, 0)
	(0, 0, -1), (-1, 0, -1)		(-2, 0, 0)	(-1, 0, 0)
M_{TurnNW}	(0, -1, 0), (0, -1, 1)	(0, 1, 0)		(0, 0, 1)
	(1, 0, 0), (1, 0, 1)	(-1, 0, 0)		(0, 0, 1)
M_T	(0, 0, 1), (1, 0, 1)	(2, 0, 0), (0, 0, 2)		(1, 0, 0)
	(1, 0, 0), (1, 0, -1)	(2, 0, 0), (0, 0, 1)		(1, 1, 0)
	(0, 0, 1), (0, 1, 1)	(1, 0, 0), (0, 0, 2)		(0, 1, 0)
	(0, 0, 1), (0, -1, 1)	(0, 0, 2), (0, -2, 0)		(0, -1, 0)
	(0, -1, 0), (0, -1, -1)	(0, 0, 1), (0, -2, 0)		(1, -1, 0)
	(0, 0, 1), (1, 0, 1)	(0, 0, 1), (0, -1, 0)		(1, 0, 0)
M_D	(0, 1, 0), (0, 1, -1)	(1, 0, 0)		(0, 0, -1)
	(0, 1, 0), (0, 1, 1)	(1, 0, 0)	(0, 0, -1)	(0, 1, -1)
	(0, 0, -1), (0, 0, -2)	(1, 0, 0)		(0, -1, -1)
	(1, 0, 0), (1, 0, -1)	(0, -1, 0)		(0, 0, -1)
	(1, 0, 0), (1, 0, 1)	(0, -1, 0)	(0, 0, -1)	(1, 0, -1)
	(0, 0, -1), (0, 0, -2)	(0, -1, 0)		(-1, 0, -1)

We prove Theorem 2 by a search algorithm.

The proposed algorithm considers planes that is parallel to the x - z plane or the y - z plane. Each plane is represented as $x = s$ for $s = 0, 1, 2, \dots$ and $y = s'$ for $s' = 0, 1, 2, \dots$. The MRS moves along each vertical line $\{(x, y, z) | x = s, y = t\}$ when it is on the plane $x = s$ for $t = 0, 1, 2, \dots$ and $\{(x, y, z) | x = t', y = s'\}$ when it is on the plane $y = s'$ for $t' = 0, 1, 2, \dots$. Figure 16 shows an execution of the proposed algorithm.

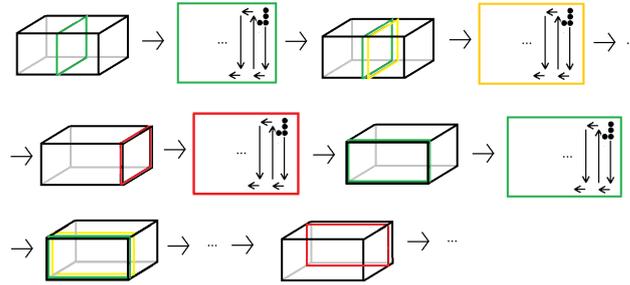
The MRS continues to search each plane perpendicular to the x -axis until it reaches the east wall. Then, it starts to search each plane perpendicular to the y -axis. Repeating the process four times, it returns to its initial position.

The proposed algorithm consists of the following move sequences. We omit the detailed description and figures due to the page limitation.

- Move sequence M_{Down} . By this move sequence, the MRS visits all the cells of the horizontal line $\{(x, y, z) | x = s, y = t\}$.
- Move sequence M_{Up} . By this move sequence, the MRS visits all the cells of the horizontal line $\{(x, y, z) | x = s, y = t\}$.
- Move sequence M_{TurnU} . By this move sequence, the MRS changes its move sequence from M_{Up} to M_{Down} .

■ **Table 2** Search algorithm for the MRS equipped with a common compass (Latter part).

	C_m	C_w	C_e	Output
M_B	$(0, 1, 0), (0, 1, 1)$	$(1, 0, 0), (0, 0, -1)$	$(0, -1, 0), (0, 2, 0)$	$(-1, 1, 0)$
	$(0, 0, -1), (-1, 0, -1)$	$(1, 0, 0), (0, 0, -2)$	$(0, -1, 0)$	$(-1, 0, 0)$
	$(1, 0, 0), (1, 0, 1)$	$(0, -1, 0), (0, 0, -1)$		$(1, 1, 0)$
	$(0, 0, -1), (0, 1, -1)$	$(0, -1, 0), (0, 0, -2)$		$(0, 1, 0)$
$M_{NECorner}$	$(0, 0, -1), (0, -1, -1)$	$(0, 1, 0), (1, 0, 0), (0, 0, -2)$		$(-1, 0, -1)$
$M_{WallBottom}$	$(1, 0, 0), (1, -1, 0)$	$(0, 0, -1)$		$(0, -1, 0)$
	$(1, 0, 0), (1, 1, 0)$	$(0, 0, -1)$	$(0, -1, 0)$	$(1, -1, 0)$
	$(0, -1, 0), (0, -2, 0)$	$(0, 0, -1)$		$(-1, -1, 0)$
	$(0, -1, 0), (-1, -1, 0)$	$(0, 0, -1), (0, -2, 0)$		$(-1, 0, 0)$
	$(0, -1, 0), (1, -1, 0)$	$(0, 0, -1)$	$(-1, 0, 0)$	$(-1, -1, 0)$
	$(-1, 0, 0), (-2, 0, 0)$	$(0, 0, -1)$		$(-1, 1, 0)$
$M_{SWCorner}$	$(-1, 0, 0), (-1, 1, 0)$	$(0, 0, -1), (-2, 0, 0), (0, -1, 0)$		$(-1, 0, 1)$
M_{Up}	$(0, -1, 0), (0, -1, 1)$	$(-1, 0, 0), (0, -2, 0)$		$(0, 0, 1)$
	$(0, -1, 0), (0, -1, -1)$	$(-1, 0, 0), (0, -2, 0)$	$(0, 0, 1)$	$(0, -1, 1)$
	$(0, 0, 1), (0, 0, 2)$	$(-1, 0, 0), (0, -1, 0)$		$(0, 1, 1)$
	Otherwise	Otherwise	Otherwise	$(0, 0, 0)$



■ **Figure 16** Example of a search by four modules.

- Move sequence M_{TurnD} . By this move sequence, the MRS changes its move sequence from M_{Down} to M_{Up} .
- Move sequence M_{B1} . By this move sequence, the MRS changes its move sequence from M_{Down} to M_{B2} .
- Move sequence M_{B2} . By this move sequence, the MRS visits all the cells of the horizontal line $\{(x, y, z) | y = s, z = 0\}$.
- Move sequence M_{B3} . By this move sequence, the MRS changes its move sequence from M_{B2} to M_{Up} .
- Move sequence M_{Corner} . By this move sequence, the MRS changes its move sequence from M_{Down} to M_{B1} .

The proposed algorithm consists of the following six steps. We use north, south, east, and west for explanation, however each module does not need to know these directions.

- Step 1** By M_{Down} , the MRS moves down along a vertical line on a plane $\{(x, y, z)|y = s\}$ for some s .
- Step 2** When the MRS reaches the bottom wall, it changes the direction by M_{TurnD} .
- Step 3** By M_{Up} , the MRS moves up along the same vertical line as Step 1.
- Step 4** If the MRS is adjacent to the bottom of west wall, it moves to the next plane $\{(x, y, z)|y = s - 1\}$ by M_{B1} . Then it moves along the bottom wall in the east direction by M_{B2} . When the MRS reaches the east wall, it performs M_{B3} and starts searching the next plane by repeating Step 1. Otherwise, it proceeds to Step 5.
- Step 5** When the MRS reaches the top wall, it moves west by one row by M_{TurnU} . Then it repeats Step 1 again.
- Step 6** When the MRS reaches southwest corner, it performs M_{Corner} and starts searching on a plane perpendicular to the previous search plane. Thus, a search plane is first perpendicular to the x axis and moves to east, second it is perpendicular to the y axis and moves to north, third it is perpendicular to the x axis and moves to west, and finally, it is perpendicular to the y axis and moves to south.
- Depending on its initial state, MRS may start from the middle of the above track.

We omit the description like Table 1 and 2 due to the page limitation.

We briefly address the correctness of the proposed algorithm. The MRS visits all cells on a vertical line $\{(x, y, z)|x = t, y = s\}$ by Steps 1, 2, and 3. Then, it proceeds to the next vertical line by Step 5. By repeating Steps 1, 2, 3, and 5, it visits all cells on plane $y = s$. By Step 4, it starts searching the next plane $y = s - 1$. By repeating Steps 1 to 5, it eventually reaches the southwest corner of the bottom wall. It starts searching the vertical line $\{(x, y, z)|x = 1, y = d - 2\}$ by Step 6. By repeating Step 1 to 6 four times, the MRS visits all cells of the field.

3.3 Search without a common compass

We show the following theorem by a search algorithm for the MRS of five modules not equipped with a common compass.

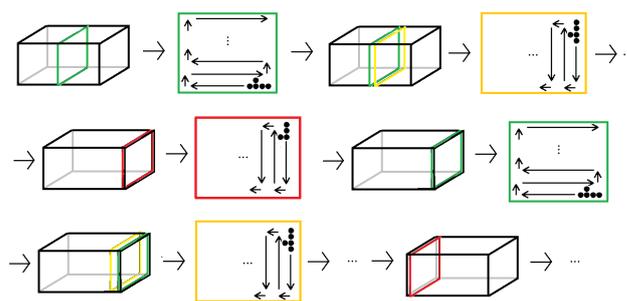
► **Theorem 3.** *The MRS of five modules not equipped with a common compass can solve a search problem in a finite 3D grid if no pair of modules have an identical observation in an initial configuration.*

We prove Theorem 3 by a search algorithm for the MRS of three modules not equipped with a common compass.

The proposed algorithm considers each plane perpendicular to one of the x , y , and z axes. The choice of the axis depends on the initial configuration of the MRS, and the modules do not need to know the global coordinate system. In the following, without loss of generality, we assume that the MRS considers planes perpendicular to the x axis, i.e., $x = s(y = s, z = s, \text{ respectively})$ for $s = 0, 1, 2, \dots$. It moves along each vertical line $\{(x, y, z)|x = s, y = t, z = u\}$ or horizontal line $\{(x, y, z)|x = s, y = u, z = t\}$ for $u = 0, 1, 2, \dots$ on the plane. Figure 17 shows an execution of the algorithm.

The MRS continues to search each plane perpendicular to the x -axis until it reaches the east wall. Then, it changes the search direction from the positive x direction to the negative x direction and it starts to search each plane perpendicular to the x -axis until it reaches the west wall.

The proposed algorithm consists of the following move sequences. We omit the detailed description and figures due to the page limitation.



■ **Figure 17** Example of a search with five modular robots.

- Move sequence $M_{Forward}$. By this move sequence, the MRS visits all the cells of the horizontal line $\{(x, y, z) | x = s, y = t\}$.
- Move sequence M_{Back} . By this move sequence, the MRS visits all the cells of the horizontal line $\{(x, y, z) | x = s, y = t\}$.
- Move sequence M_{TurnB} . By this move sequence, the MRS changes its move sequence from $M_{Forward}$ to M_{Back} .
- Move sequence M_{TurnF} . By this move sequence, the MRS changes its move sequence from M_{Back} to $M_{Forward}$.
- Move sequence M_{Edge} . By this move sequence, the MRS changes its move sequence from $M_{Forward}$ to M_{TurnB} .
- Move sequence M_{Corner} . By this move sequence, the MRS changes its move sequence from $M_{Forward}$ to M_{TurnB} .

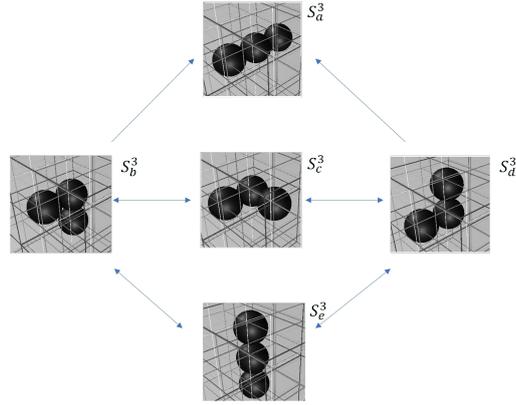
The proposed algorithm consists of the following six steps. We use down direction for explanation, but each module does not need to know the down direction.

- Step 1** By $M_{Forward}$, the MRS moves to the down direction along a vertical line on a plane $\{(x, y, z) | x = s\}$ for some s .
- Step 2** When the MRS reaches the bottom wall, it changes the direction to up by M_{TurnB} .
- Step 3** By M_{Back} , the MRS moves up along a vertical line followed in Step 1.
- Step 4** If the MRS is adjacent to the bottom wall, it moves to plane $\{(x, y, z) | x = s + 1\}$ by M_{TurnF} , and starts searching the new plane by repeating Steps 1, 2, and 3.
- Step 5** When the MRS reaches the top or bottom wall, it moves to the southern row by M_{Edge} . Then it repeats Steps 1, 2, and 3 again.
- Step 6** When the MRS reaches a corner of the field, it perform M_{Corner} and changes the search direction from the positive x direction to the negative x direction.

We omit the description like Table 1 and 2 due to the page limitation.

We briefly address the correctness of the proposed algorithm. The MRS visits all the cells on line $\{(x, y, z) | x = s, y = t\}$ by Steps 1, 2, and 3. Then, it proceeds to the next line by Step 5. By repeating Steps 1, 2, 3, and 5, it visits all cells on plane $x = s$. By Step 4, it starts searching a new plane $x = s + 1$. By repeating Steps 1 to 5, it eventually reaches the corner adjacent to the south wall and the east wall. By Step 6, it starts searching west wall. By repeating Steps 1 to 6 twice, the MRS visits all cells of the field.

You can find demonstration videos of the proposed algorithms in [18].



■ **Figure 18** State transition graph for a MRS consisting of 3 modules with common vertical axis.

4 Necessary number of modules

We show that the three algorithms presented in Section 3 use the minimum number of modules for each setting.

► **Theorem 4.** *The MRS of less than three modules equipped with a common compass cannot solve the search problem in a finite 3D cubic grid.*

Due to the page limitation, we show a sketch of the proof.

A single module cannot perform any movement because there is no static module during the movement.

Two modules can perform rotations and we can show that the MRS can move straight to one direction by repeating rotations. Thus, when both modules cannot observe any wall (i.e., in the middle area of the field), the MRS moves straight. Assume that the straight movement of the MRS is parallel to x axis. When the field is large enough, the MRS cannot move along some lines parallel to the x axis because the MRS cannot count.

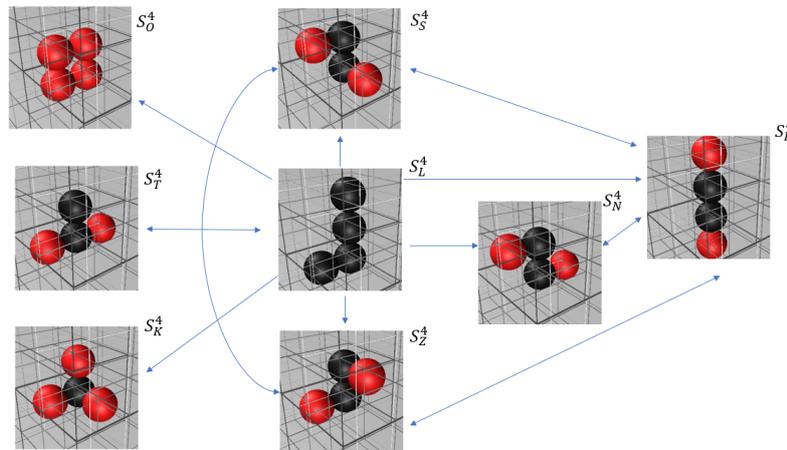
We next show the necessary number of modules equipped with a common vertical axis.

► **Theorem 5.** *The MRS of less than four modules equipped with a common vertical axis cannot solve the search problem in a finite 3D cubic grid.*

Proof. In the case of one module, the MRS cannot move because it cannot perform any sliding or rotation.

In the case of two modules, there are two possible states of the MRS. Let S_A be the state, where the two modules form a vertical line, and S_B be the state, where the two modules form a horizontal line. There exists only one horizontal line state because the modules do not agree on x axis or y axis. In S_A , one of the two modules can perform a rotation because they agree on a common vertical axis. Any rotation in S_A results in S_B . In S_B , both modules obtain the same observation if their local coordinate systems are symmetric against their midpoint, and if one of them moves then the other also moves. Thus, the two modules cannot perform any movement. Consequently, the two modules cannot move to any direction.

In the case of three modules, we check possible movements of the MRS by the state transition graph shown in Figure 18. State S_a^3 cannot be transformed to any other configuration because both endpoint modules obtain the same observation. Therefore, it is necessary to move only by S_b^3, S_c^3, S_d^3 , and S_e^3 . However, no matter which transformation of the S_b^3, S_c^3, S_d^3 , and S_e^3 , the MRS cannot move in the east, west, south or south direction. Therefore, when there is no wall in the visibility, the MRS cannot move, thus it cannot search the whole field. ◀



■ **Figure 19** State transition graph for the MRS of 4 modules not equipped with a common compass.

We finally show the necessary number of modules not equipped with a common compass.

► **Theorem 6.** *The MRS of less than five modules not equipped with a common compass cannot solve the search problem in a finite 3D cubic grid.*

Due to the page limitation, we show a sketch of the proof.

A single module cannot perform any movement because there is no static module during the movement.

Two modules not equipped with a common compass cannot perform any movement, because if one module moves then the other module also moves.

Three modules have two states, i.e., the “L”-shape and the “I”-shape. In the L-shape, no module can perform a movement because of the symmetry. In the I-shape, two endpoint modules can perform rotations and a new state is an L-shape or I-shape. The MRS does not move after these rotations. Consequently, the MRS cannot move to any direction.

Four modules have eight states. By the state transition graph of the four modules shown in Figure 19, we can show that the MRS cannot move to any direction.

5 Conclusion and future work

In this paper, we considered search by the single MRS in the finite 3D cubic grid. We demonstrated a trade-off between the common knowledge and the necessary and sufficient number of modules for search. We finally note that the proposed algorithms depend on parallel movements, i.e., they are not designed for the centralized scheduler.

Our future goal is a distributed coordination theory for the MRS. First, reconfiguration and locomotion of a single MRS in the 3D cubic grid have not been discussed yet. Second, it is important to consider interaction among multiple MRSs such as rendezvous, collision avoidance, and collective search. Finally, the MRS is expected to solve more complicated tasks by interaction with the environment.

References

- 1 Abdullah Almethen, Othon Michail, and Igor Potapov. On efficient connectivity-preserving transformations in a grid. In *Proc. of ALGOSENSORS 2020*, pages 76–91, 2020.
- 2 Abdullah Almethen, Othon Michail, and Igor Potapov. Pushing lines helps: Efficient universal centralised transformations for programmable matter. *Theoretical Computer Science*, 830-831:43–59, 2020.
- 3 Dana Angluin, Zoë Diamadi James Aspnes, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. In *Proc. of PODC 2004*, pages 290–299, 2004.
- 4 Shantanu Das, Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Masafumi Yamashita. Autonomous mobile robots with lights. *Theoretical Computer Science*, 609:171–184, 2016.
- 5 Zahra Derakhshandeh, Shlomi Dolev, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Brief announcement: amoebot - a new model for programmable matter. In *Proc. of SPAA 2014*, pages 220–222, 2014.
- 6 Zahra Derakhshandeh, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. An algorithmic framework for shape formation problems in self-organizing particle systems. In *Proc. of NANOCOM 2015*, pages 21:1–21:2, 2015.
- 7 Keisuke Doi, Yukiko Yamauchi, Shuji Kijima, and Masafumi Yamashita. Exploration of finite 2d square grid by a metamorphic robotic system. In *Proc. of SSS 2018*, pages 96–110, 2018.
- 8 Adrian Dumitrescu and János Pach. Pushing squares around. *Graphs and Combinatorics*, 22:37–50, 2006.
- 9 Adrian Dumitrescu, Ichiro Suzuki, and Masafumi Yamashita. Formations for fast locomotion of metamorphic robotic systems. *The International Journal of Robotics Research*, 23(6):583–593, 2004.
- 10 Adrian Dumitrescu, Ichiro Suzuki, and Masafumi Yamashita. Motion planning for metamorphic systems: feasibility, decidability, and distributed reconfiguration. *IEEE Transactions on Robotics*, 20(3):409–418, 2004.
- 11 Nao Fujinaga, Yukiko Yamauchi, Hirotaka Ono, Shuji Kijima, and Masafumi Yamashita. Pattern formation by oblivious asynchronous mobile robots. *SIAM Journal on Computing*, 44(3):740–785, 2015.
- 12 Giuseppe Antonio Di Luna, Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Yukiko Yamauchi. Shape formation by programmable particles. *Distributed Computing*, 33(1):69–101, 2020.
- 13 Othon Michail, George Skretas, and Paul G. Spirakis. On the transformation capability of feasible mechanisms for programmable matter. *Journal of Computer and System Sciences*, 102:18–39, 2019.
- 14 Junya Nakamura, Sayaka Kamei, and Yukiko Yamauchi. Evacuation from a finite 2d square grid field by a metamorphic robotic system. In *Proc. of CANDAR 2020*, pages 69–78, 2020.
- 15 Michael Rubenstein, Alejandro Cornejo, and Radhika Nagpal. Programmable self-assembly in a thousand-robot swarm. *Science*, 345(6198):795–799, 2014.
- 16 Ichiro Suzuki and Masafumi Yamashita. Distributed anonymous mobile robots: Formation of geometric patterns. *SIAM Journal on Computing*, 28(4):1347–1363, 1999.
- 17 Pierre Thalamy, Benoît Piranda, and Julien Bourgeois. Distributed self-reconfiguration using a deterministic autonomous scaffolding structure. In *Proc. of AAMAS 2019*, pages 140–148, 2019.
- 18 Ryonosuke Yamada. MRS demonstration videos. URL: <http://tcs.inf.kyushu-u.ac.jp/~yamauchi/MRSdemonstrations.html>.

Brief Announcement: Cooperative Guarding in Polygons with Holes

John Augustine   

Department of Computer Science & Engineering, Indian Institute of Technology Madras, India

Srikanth Ramachandran  

Department of Computer Science & Engineering, Indian Institute of Technology Madras, India

Abstract

We study the Cooperative Guarding problem for polygons with holes in a mobile multi-agents setting. Given a set of agents, initially deployed at a point in a polygon with n vertices and h holes, we require the agents to collaboratively explore and position themselves in such a way that every point in the polygon is visible to at least one agent and that the set of agents are visibly connected. We study the problem under two models of computation, one in which the agents can compute exact distances and angles between two points in its visibility, and one in which agents can only compare distances and angles. In the stronger model, we provide a deterministic $O(n)$ round algorithm to compute such a cooperative guard set while not requiring more than $\frac{n+h}{2}$ agents and $O(\log n)$ bits of persistent memory per agent. In the weaker model, we provide an $O(n^4)$ round algorithm, that does not require more than $\frac{n+2h}{2}$ agents.

2012 ACM Subject Classification Computing methodologies → Self-organization

Keywords and phrases Mobile Agents, Art Gallery Problem, Cooperative Guarding

Digital Object Identifier 10.4230/LIPIcs.SAND.2022.21

Related Version *Full Version:* <https://arxiv.org/abs/2202.13719>

Funding *John Augustine:* DST/SERB MATRICS under Grant MTR/2018/001198; Exploratory Research Grant at IIT Madras; Cryptography, Cybersecurity, & Distributed Trust prospective Centre of Excellence under the Institute of Eminence Scheme.

Srikanth Ramachandran: Exploratory Research Grant at IIT Madras; Cryptography, Cybersecurity, & Distributed Trust prospective Centre of Excellence under the Institute of Eminence Scheme.

Acknowledgements We thank Barath Ashok and Suman Sourav for helpful discussions and the anonymous reviewers for their useful feedback.

1 Introduction

The *Art Gallery Problem* is a classical computational geometry problem which asks for the minimum number of guards required to completely guard the interior of a given art gallery. The art gallery is modelled as a simple polygon and guards are points on or inside the polygon. A set of guards is said to guard the art gallery, if every point in the art gallery is visible to at least one guard. This problem was first posed by Klee in 1973 and since then the problem has been of interest to researchers. The problem and its many variations have been well-studied over the years. Chvátal [4] was the first to show that $\lfloor \frac{n}{3} \rfloor$ guards are sufficient and sometimes necessary to guard a polygon with n vertices. Fisk [5] proved the same result via an elegant coloring argument, which also leads to an $O(n)$ algorithm.

We study a variant of the classical Art Gallery Problem known as the *Cooperative Guards* problem in polygons with holes from a distributed multi-agents perspective. The *Cooperative Guards* problem is similar to the *Art Gallery Problem* with the additional constraint that the visibility graph of the guards, i.e., the graph with guards as vertices and edges between guards that can see each other, should form a single connected component. This implies that if the guards can communicate through line of sight, then any two guards can communicate



© John Augustine and Srikanth Ramachandran;
licensed under Creative Commons License CC-BY 4.0

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).

Editors: James Aspnes and Othon Michail; Article No. 21; pp. 21:1–21:3

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

with each other (either directly or indirectly through intermediate guards). The *Cooperative Guards* problem was first introduced by Liaw *et al.* in 1993 [6], in which they show that this variant of the *Art Gallery Problem* is also NP-hard.

The pursuit of solving the *Cooperative Guards* problem for us is primarily motivated by the search for distributed multi-agent exploration algorithms for agents deployed in the polygon. One of the easiest ways to explore the polygon is by maintaining connectivity through line of sight and having the agents cooperatively send exploratory agents and expand the area of the polygon that they can collectively see. Such an algorithm would require the agents to be visibly connected and if at the end they are collectively able to see the entire polygon, then they form a solution to the *Cooperative Guards* problem. This led us to first tackle the *Cooperative Guards* problem in the centralized setting and adapt it to the distributed multi-agent model. Once the agents are positioned to cooperatively guard the polygon, they have sufficient information to solve several computational geometry problems on the polygon by executing a distributed algorithm on their visibility graph. Hence our algorithm can also serve as an initial pre-processing step for distributed multi-agents to solve other problems on the polygon, for example they can compute the number of vertices of the polygon, diameter of the polygon etc. Our focus throughout this paper is to find time optimal algorithms for computing a set of cooperative guards on the polygon (not necessarily minimum). Analogous to Fisk's [5] coloring argument, we provide an efficient algorithm that computes a set of visibly connected guards using no more than $\frac{n+h-2}{2}$ guards.

2 Our Contributions

We first present a centralized algorithm for the Cooperative Guards problem in polygonal region with n vertices and h holes, that does not require more than $\frac{n+2h-2}{2}$ guards and runs in $O(n \log n)$ time. We then use an observation from Zylinski [8] to reduce the number of guards to $\frac{n+h-2}{2}$. The reduction uses the method of Sachs and Souvaine [3] to reduce the problem to a simple polygon with $n + h$ vertices and no holes.

We propose two distributed models of computation, one in which the agents can perceive exact distances (which we call *depth perception*) and hence can compute co-ordinates (with respect to some common reference frame) to map out the polygon, and another in which the agents receive only a combinatorial view of their visibility. In the former model we assume that the agents are positioned in an $[0, n^c] \times [0, n^c]$ grid so that their coordinates can be evaluated within $O(\log n)$ bits.

We present a distributed algorithm (in the stronger *depth perception* model) that runs in $O(n)$ rounds and does not require more than $\lfloor \frac{n+h-2}{2} \rfloor$ agents. Our distributed algorithm emulates the centralized algorithms presented except that the polygon is not known in advance. We simultaneously explore and incrementally construct a triangulation.

Finally, we present a distributed algorithm for the weaker *proximity perception* model, which takes $O(n^4)$ rounds, but requires $\frac{n+2h-2}{2}$ agents. These results improve upon the works by Obermeyer, Ganguli and Buffon [7] (which requires $\frac{n+2h-2}{2}$ guards in the worst case and $O(n^2)$ communication rounds) and Ashok et al. [1] (which requires $O(n)$ rounds but works only for polygons without holes). A summary of our results is presented in Table 1. The full version [2] of the paper contains all the details.

3 Open Problems

We discuss about further improvements and some closely related open problems below.

■ **Table 1** A summary of our results.

Model & Algorithm	Rounds	Broadcasts	Persistent Memory	Guards
Depth Perception (Large memory)	$O(n)$	$O(n)$	$O(n \log n)$	$(n + h - 2)/2$
Depth Perception (Small memory)	$O(n)$	$O(n \log n)$	$O(\log n)$	$(n + 2h - 2)/2$
Depth Perception (Improving guards)	$O(n)$	$O(n(h + \log n))$	$O(\log n)$	$(n + h - 2)/2$
Proximity Perception	$O(n^4)$	$O(n^4)$	$O(\log n)$	$(n + 2h - 2)/2$

► **Open Problem 1.** *How many co-operative guards are required for a polygon with n vertices and h holes? What if guards are restricted to be on the vertices of the polygon?*

Our construction requires at most $\frac{n+h-2}{2}$ co-operative guards, however we have not been able to construct polygons requiring so many guards. For $h \leq 1$, the problem has been solved by Zylinski [8], but for $h \geq 2$, the problem remains open. We have not been able to provide a better upper bound than $\frac{n+2h-2}{2}$ guards when the guards are restricted to the vertices of the polygon as well as provide examples of polygons that need more than $\frac{n+h-2}{2}$ such guards.

► **Open Problem 2.** *Is there a more efficient centralized algorithm to compute a cooperative guard set with $\frac{n+h-2}{2}$ guards?*

The centralized algorithm proposed runs in $O(n^2)$ time, whereas we are able to construct a cooperative guard set using no more than $\frac{n+2h-2}{2}$ agents in $O(n \log n)$ time. Is it possible to bridge the gap in the running time?

► **Open Problem 3.** *Is it possible to construct $O(\text{poly}(D))$ time algorithms where D is the hop diameter of the polygon, when agents have limited persistent memory?*

The hop diameter of the polygon is the minimum number of straight line segments required to connect any two vertices of the polygon. In simple polygons, it is easy to construct such algorithms, however in polygons with holes, the problem appears to be more challenging.

References

- 1 B. Ashok, J. Augustine, A. Mehekare, S. Ragupathi, S. Ramachandran, and S. Sourav. Guarding a polygon without losing touch. In *SIROCCO*, pages 91–108, 2020.
- 2 J. Augustine and S. Ramachandran. Cooperative guarding in polygons with holes. [arXiv:2202.13719](https://arxiv.org/abs/2202.13719).
- 3 I. Bjorling-Sachs and D. L. Souvaine. An efficient algorithm for guard placement in polygons with holes. *Discrete & Computational Geometry*, 13(1):77–109, January 1995. doi:10.1007/bf02574029.
- 4 V. Chvátal. A combinatorial theorem in plane geometry. *Journal of Combinatorial Theory, Series B*, 18(1):39–41, 1975.
- 5 Steve Fisk. A short proof of chvátal’s watchman theorem. *Journal of Combinatorial Theory, Series B*, 24(3):374, 1978.
- 6 B. Liaw, N. Huang, and Richard C. T. Lee. The minimum cooperative guards problem on k-spiral polygons. In *CCCG*, 1993.
- 7 K. J. Obermeyer, A. Ganguli, and F. Bullo. Multi-agent deployment for visibility coverage in polygonal environments with holes. *International Journal of Robust and Nonlinear Control*, 21(12):1467–1492, 2011.
- 8 P. Zylinski. Cooperative guards in art galleries with one hole. *Balkan Journal of Geometry and Its Applications*, 10(2):142, 2005.

Brief Announcement: The Temporal Firefighter Problem

Samuel D. Hand ✉ 

School of Computing Science, University of Glasgow, UK

Jessica Enright ✉ 

School of Computing Science, University of Glasgow, UK

Kitty Meeks ✉ 

School of Computing Science, University of Glasgow, UK

Abstract

The FIREFIGHTER problem asks how many vertices can be saved from a fire spreading over the vertices of a graph. At timestep 0 a vertex begins burning, then on each subsequent timestep a non-burning vertex is chosen to be defended, and the fire then spreads to all undefended vertices that it neighbours. The problem is NP-Complete on arbitrary graphs, however existing work has found several graph classes for which there are polynomial time solutions. We introduce TEMPORAL FIREFIGHTER, an extension of FIREFIGHTER to temporal graphs. We show that TEMPORAL FIREFIGHTER is also NP-Complete, and remains so on all but one of the underlying classes of graphs on which FIREFIGHTER is known to have a polynomial-time solution. This motivates us to explore restrictions on the temporal structure of the graph, and we find that TEMPORAL FIREFIGHTER is fixed parameter tractable with respect to the temporal graph parameter vertex-interval-membership-width.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases Temporal graphs, Spreading processes, Parameterised complexity

Digital Object Identifier 10.4230/LIPIcs.SAND.2022.22

Related Version *Full Version*: <https://arxiv.org/abs/2202.12599> [4]

Funding *Samuel D. Hand*: Supported by an EPSRC doctoral training account.

Jessica Enright: Supported by EPSRC grant EP/T004878/1.

Kitty Meeks: Supported by EPSRC grant EP/T004878/1.

1 Problem Definition and Restrictions on the Underlying Graph

The FIREFIGHTER problem considers a fire spreading over a connected, undirected, loop-free, rooted graph [5]. At time 0 the fire begins burning at the root. Then, at each subsequent time, a chosen vertex is defended, and the fire then spreads to all undefended vertices adjacent to the fire. Once a vertex is burning or defended it remains so until the process is over, which happens once the fire can no longer spread. The decision problem then asks whether it is possible to save k vertices on a rooted graph (G, r) . This problem is NP-Complete on arbitrary graphs, although progress has been made on identifying graph classes for which it can be solved in polynomial time [2, 3].

We introduce TEMPORAL FIREFIGHTER, an extension of FIREFIGHTER to temporal graphs, using the definition of temporal graph first introduced by Kempe et al. [6]. Here a temporal graph is a pair (G, λ) , where G is an underlying static graph and $\lambda : E(G) \rightarrow 2^{\mathbb{N}}$ is a labeling function mapping edges of the graph to the set of timesteps at which they are active. We refer to the maximum timestep at which any edge of such a graph is active as the lifetime Λ . Furthermore, we say that two vertices v_1 and v_2 are temporally adjacent at time i if there is an edge between them active at time i , that is if $i \in \lambda(v_1, v_2)$. TEMPORAL FIREFIGHTER is then defined analogously to FIREFIGHTER except the fire can only spread to vertices to which it is temporally adjacent.



© Samuel D. Hand, Jessica Enright, and Kitty Meeks;
licensed under Creative Commons License CC-BY 4.0

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).

Editors: James Aspnes and Othon Michail; Article No. 22; pp. 22:1–22:3

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

TEMPORAL FIREFIGHTER is NP-Complete on arbitrary graphs, as we can assign times in a rooted temporal graph $((G, \lambda), r)$ such that TEMPORAL FIREFIGHTER simulates FIREFIGHTER for any rooted graph (G, r) . This is achieved by setting $\lambda(e) = \{1, \dots, |V(G)| - 1\}$ for every edge e . By time $|V(G)| - 1$ every vertex would have been defended, so the process must be over. Thus, for the entirety of the time during which the fire can spread, every edge is active, just as in FIREFIGHTER. In this respect we can view FIREFIGHTER to be a special case of TEMPORAL FIREFIGHTER. Noting that the above reduction preserves the underlying graph class we then have that for every class \mathcal{C} of graphs for which FIREFIGHTER is NP-Complete, TEMPORAL FIREFIGHTER is NP-Complete on the class of temporal graphs with the graphs of \mathcal{C} as the underlying graphs.

FIREFIGHTER has polynomial time solutions on interval graphs, permutation graphs, P_k -free graphs for $k > 5$, split graphs, cographs, and graphs of maximum degree three providing the root is of degree two [2, 3]. We find that TEMPORAL FIREFIGHTER is NP-Complete when the underlying graph belongs to any of these classes except the last, for which there is a polynomial time algorithm. Every class except the last contains arbitrarily large cliques, and we find that TEMPORAL FIREFIGHTER is NP-Complete when the underlying graph is a clique. This can be shown by reduction from FIREFIGHTER by assigning times to the edges in a static graph G so that they will be active at all times up until $|V(G)| - 1$, at which point the fire can certainly no longer spread. We then add further edges to make the graph a clique, and have them only active from time $|V(G)| - 1$ onwards such that they will not affect the spread of the fire. TEMPORAL FIREFIGHTER on such a clique will then simulate FIREFIGHTER on G .

We also find the stronger result, that TEMPORAL FIREFIGHTER is NP-Complete when the underlying graph is a clique of size n , even when the lifetime of the graph is upper bounded by $n^{\frac{1}{c}}$ for any constant $c \in \mathbb{N}$. This can be shown by a similar reduction, in which $|G|^c - |G|$ vertices are added to the static graph G in such a way that they will all burn on the first timestep. All defences then take place on a clique constructed in the manner described above.

In the positive direction we find that there is a polynomial time algorithm for TEMPORAL FIREFIGHTER on temporal graphs where the underlying graph has maximum degree three and the root has degree two. This algorithm works in a very similar manner to that for FIREFIGHTER as given by Finbow et al. [2].

2 Restrictions on the Temporal Structure

Our analysis of the complexity of TEMPORAL FIREFIGHTER when restricting the underlying graph class shows that for most known graph classes \mathcal{C} where FIREFIGHTER is polytime solvable, TEMPORAL FIREFIGHTER is NP-Complete on the class of temporal graphs $\{(G, \lambda) : G \in \mathcal{C}\}$. This naturally leads us to consider whether the problem might be tractable when restricting the temporal structure of the graph.

We show that TEMPORAL FIREFIGHTER is fixed parameter tractable when parameterised by vertex-interval-membership-width, a temporal graph parameter defined by Bumpus and Meeks [1]. Begin by letting $\text{mintime}(v)$ denote the minimum timestep upon which an incident edge of v is active for all vertices v . Define maxtime equivalently for the maximum timestep. Vertex-interval-membership-width is then defined as follows.

► **Definition 1** (Vertex Interval Membership Width). *The vertex interval membership sequence of a temporal graph (G, λ) is the sequence $(F_t)_{t \in [\Lambda]}$ of vertex-subsets of G where $F_t = \{v \in V(G) : \text{mintime}(v) \leq t \leq \text{maxtime}(v)\}$ and Λ is the lifetime of (G, λ) . The vertex-interval-membership-width of a temporal graph (G, λ) is then the integer $\omega = \max_{t \in [\Lambda]} |F_t|$.*

To simplify our analysis in showing that TEMPORAL FIREFIGHTER is in FPT, we actually use the related problem TEMPORAL FIREFIGHTER RESERVE. This is the temporal extension of the FIREFIGHTER RESERVE problem described by Fomin et al. [3]. In TEMPORAL FIREFIGHTER RESERVE, it is not required to make a defensive move every timestep. Rather, defences may be delayed, adding to a budget that can be used on future timesteps. Allowing the defence to build up a reserve in this manner does not affect the number of vertices that can be saved, and the proof for this fact works identically to that for the static case as given by Fomin et al. [3]. We note that in TEMPORAL FIREFIGHTER RESERVE there is always an optimal strategy that only defends temporally adjacent to the fire, as any defence can be delayed until the turn upon which the defended vertex would burn.

The algorithm for TEMPORAL FIREFIGHTER RESERVE iterates over the vertex interval membership sequence of the input graph, and considers every possible set of defences adjacent to the fire for each timestep. In particular it recursively computes a sequence of sets $L_i \in \mathcal{P}(F_i) \times \mathcal{P}(F_i) \times \{1, 2, \dots, \Lambda\} \times \{1, 2, \dots, n\}$. An element of L_i is a 4-tuple (D, B, g, c) where D is a set of defended vertices in F_i , B is a set of burnt vertices in F_i , g is the budget that will be available on timestep $i + 1$, and c is the total count of vertices that have burnt at time i . The problem can then be answered by checking if there is any entry $(D, B, g, c) \in L_\Lambda$ where Λ is the lifetime of the graph, such that $|V(G)| - c \geq k$.

We observe that $|L_i| = O(4^\omega \omega \Lambda^2)$, and that there at most 2^ω defences to consider on each timestep. The overall complexity can then be seen to be $O(8^\omega \omega \Lambda^3)$.

► **Theorem 2.** *It is possible to solve TEMPORAL FIREFIGHTER in time $O(8^\omega \omega \Lambda^3)$ for a rooted temporal graph $((G, \lambda), r)$ where Λ is the lifetime of the graph, and ω is the vertex-interval-membership-width.*

We suggest investigating the complexity of TEMPORAL FIREFIGHTER when parameterised by similar temporal parameters as future work. For example, it would be worthwhile investigating parameterising by interval-membership-width, which is also defined by Bumpus and Meeks [1].

References

- 1 Benjamin Merlin Bumpus and Kitty Meeks. Edge exploration of temporal graphs. *CoRR*, abs/2103.05387, 2021. [arXiv:2103.05387](#).
- 2 Stephen Finbow, Andrew D. King, Gary MacGillivray, and Romeo Rizzi. The firefighter problem for graphs of maximum degree three. *Discret. Math.*, 307(16):2094–2105, 2007. [doi:10.1016/j.disc.2005.12.053](#).
- 3 Fedor V. Fomin, Pinar Heggernes, and Erik Jan van Leeuwen. The firefighter problem on graph classes. *Theor. Comput. Sci.*, 613:38–50, 2016. [doi:10.1016/j.tcs.2015.11.024](#).
- 4 Samuel Hand, Jessica Enright, and Kitty Meeks. Making life more confusing for firefighters, 2022. [arXiv:2202.12599](#).
- 5 Bert Hartnell. Firefighter! an application of domination. In *the 24th Manitoba Conference on Combinatorial Mathematics and Computing, University of Minitoba, Winnipeg, Cadada, 1995*, 1995.
- 6 David Kempe, Jon M. Kleinberg, and Amit Kumar. Connectivity and inference problems for temporal networks. *J. Comput. Syst. Sci.*, 64(4):820–842, 2002. [doi:10.1006/jcss.2002.1829](#).

Brief Announcement: Fault-Tolerant Shape Formation in the Amoebot Model

Irina Kostitsyna ✉ 

Department of Mathematics and Computer Science,
Eindhoven University of Technology, The Netherlands

Christian Scheideler ✉ 

Department of Computer Science, Paderborn University, Germany

Daniel Warner ✉ 

Department of Computer Science, Paderborn University, Germany

Abstract

The amoebot model is a distributed computing model of programmable matter. It envisions programmable matter as a collection of computational units called amoebots or particles that utilize local interactions to achieve tasks of coordination, movement and conformation. In the geometric amoebot model the particles operate on a hexagonal tessellation of the plane. Within this model, numerous problems such as leader election, shape formation or object coating have been studied. One area that has not received much attention so far, but is highly relevant for a practical implementation of programmable matter, is fault tolerance. The existing literature on that aspect allows particles to crash but assumes that crashed particles do not recover. We propose a new model in which a crash causes the memory of a particle to be reset and a crashed particle can detect that it has crashed and try to recover using its local information and communication capabilities. We propose an algorithm that solves the hexagon shape formation problem in our model if a finite number of crashes occur and a designated leader particle does not fail. At the heart of our solution lies a fault-tolerant implementation of the spanning forest primitive, which, since other algorithms in the amoebot model also make use of it, is also of general interest.

2012 ACM Subject Classification General and reference → General conference proceedings

Keywords and phrases Programmable matter, Geometric amoebot model, Fault tolerance, Shape formation

Digital Object Identifier 10.4230/LIPIcs.SAND.2022.23

1 Introduction

Model extension

In our work we extend the amoebot model by introducing *particle crashes*. In order to gain initial insights into useful strategies towards fault tolerance in our model and motivate further work in this direction, we focus on the problem of shape formation in the geometric amoebot model using the hexagon shape formation problem as basis. We assume that the adversarial scheduler may arbitrarily crash particles. A crash of a particle p has the following effects: The scheduler sets the *state* in p 's local memory to `CRASHED`, enabling p and its neighbours to detect that it has crashed, and resets the rest of p 's local memory. The faulty particle p can then try to recover its local memory by using its local information and communication capabilities.

Problem description

For any two nodes $u, v \in V_\Delta$ of the triangular lattice G_Δ the *distance* $\delta(u, v) \in \mathbb{N}_0$ between u and v is defined as the length of a shortest path from u to v in G_Δ . For a node $v \in V_\Delta$ and $i \in \mathbb{N}_0$ let $B(v, i) := \{u \in V_\Delta \mid \delta(u, v) = i\}$. We call a set $V \subseteq V_\Delta$ a *hexagon with centre* $v \in V$ if there is a $k \in \mathbb{N}_0$ and a subset $S \subseteq B(v, k)$ such that $V = S \cup \bigcup_{i < k} B(v, i)$. We



© Irina Kostitsyna, Christian Scheideler, and Daniel Warner;
licensed under Creative Commons License CC-BY 4.0

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).

Editors: James Aspnes and Othon Michail; Article No. 23; pp. 23:1–23:3

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

define the *hexagon shape formation problem* **HEX**: We assume that the system of particles initially forms a single connected component of contracted particles, has a unique leader, called the **SEED** particle, and that all other particles are **IDLE**. The goal is to reach a stable configuration in which the set of nodes occupied by particles is a hexagon with the **SEED** in its centre.

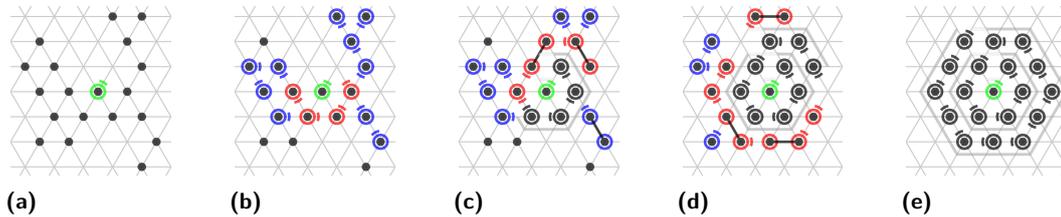
2 Main results

We propose an algorithm **HexagonFT** that solves the hexagon shape formation problem **HEX** in our model under the presence of particle crashes. Our two main results are:

► **Lemma 1.** *If a finite number of crashes occur during the execution of algorithm **HexagonFT** and m particles are faulty after the last crash, then a non-faulty configuration is reached within $\mathcal{O}(mn)$ rounds after the last crash.*

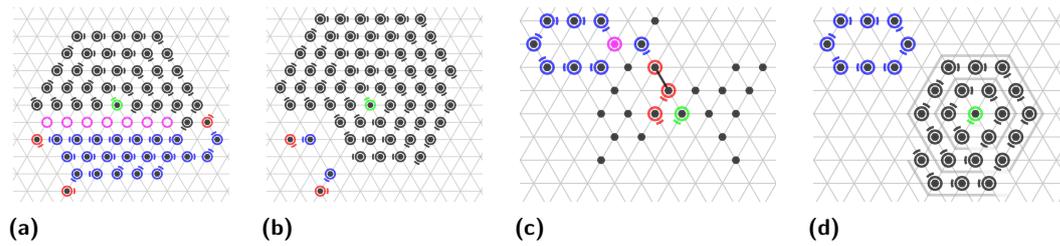
► **Theorem 2.** *If a finite number of crashes occur, then the algorithm **HexagonFT** solves the hexagon shape formation problem **HEX** in worst-case $\mathcal{O}(n^2)$ work (total number of moves executed by all particles). From the time when no more crashes occur and the configuration is non-faulty, the algorithm needs $\mathcal{O}(n)$ rounds until termination.*

As long as no crashes occur, **HexagonFT** behaves like the classical hexagon shape formation algorithm introduced in [1] (compare Figure 1).

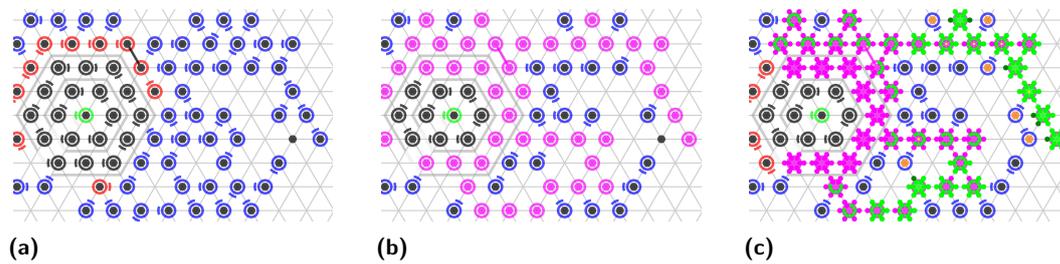


■ **Figure 1** An example run of our hexagon shape formation algorithm **HexagonFT** with 19 particles without crashes. Particles have to assume the shape of a hexagon (but for the outer layer, which may not be completely full). The hexagon is built in a spiral ring in clockwise direction around the **SEED** as follows: (a) All particles except of the **SEED** are initially **IDLE** (black dots). (b) Particles adjacent to *finished* particles (**SEED** or **RETIRED**) become **ROOT** particles, and **FOLLOWER** particles form parent-child relationships with **ROOT** or **FOLLOWER** particles. (c)–(e) **ROOT** particles traverse the forming hexagon counter-clockwise, becoming **RETIRED** when reaching the position marked by the last **RETIRED** particle. **FOLLOWER** particles follow **ROOT** particles via a series of handovers.

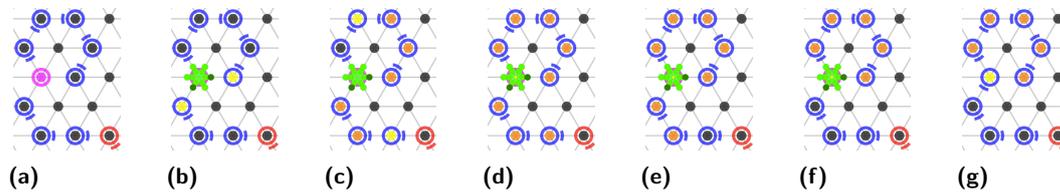
Due to space limitations, we address two algorithmic challenges that arise due to particle crashes (Figure 2): Firstly, we must ensure that particles within the hexagon formed so far do not become **FOLLOWER** particles. If this is not ensured, particles could leave the hexagon, which in turn could lead to disconnection of the particles. We use a *safety primitive* (Figure 3) to ensure that particles inside the hexagon cannot become **FOLLOWER** particles. Secondly, we need to ensure that when a crashed particle chooses a **FOLLOWER** as parent, this does not lead to disconnection of the particles. In order to avoid disconnection, we use a *validation primitive* (Figure 4) that determines for a faulty particle which of the **FOLLOWER** parent candidates it can attach to without closing a cycle.



■ **Figure 2** (a)–(b) Crashed particles inside the hexagon have become followers. Some of these followers follow their root, causing them to leave the hexagon, which eventually leads to a disconnection of the particles. (c)–(d) A **Crashed** particle attaches itself to an arbitrary FOLLOWER pointing away from it, closing a cycle and leading to irreversible disconnection of the particles.



■ **Figure 3** *Safety primitive*: Crashed particles will become either **SAFE** or **UNSAFE**. Crashed particles connected to a finished particle via one or two line segments in G_{Δ} become **UNSAFE**, otherwise **SAFE** by the propagation of *safeFlags*. Only a **SAFE** particle may become a FOLLOWER.



■ **Figure 4** *Validation primitive*: (a) A faulty particle needs to ensure that there is a path to a ROOT before following a particle. (b) The faulty particle sends **INVALIDATE** tokens to possible parent candidates. (c) **INVALIDATE** is propagated upwards, causing particles on the path to become **INVALID**. (d)–(e) An **INVALIDATE** that reaches an **ERROR** particle is stored by it, an **INVALIDATE** that reaches a **ROOT** is consumed by it. The **ROOT** generates a **VALID** token which is propagated downwards along the **INVALID** particles, causing them to become **VALID**. (f)–(g) A **SAFE** particle may become a FOLLOWER of a **VALID** FOLLOWER parent candidate. After the recovery of the particle, the **INVALIDATE** token previously stored by it will then again be propagated upwards.

References

1 Zahra Derakhshandeh, Robert Gmyr, Andréa W Richa, Christian Scheideler, and Thim Strothmann. An algorithmic framework for shape formation problems in self-organizing particle systems. In *Proceedings of the Second Annual International Conference on Nanoscale Computing and Communication*, pages 1–2, 2015.

Brief Announcement: Barrier-1 Reachability for Thermodynamic Binding Networks Is PSPACE-Complete

Austin Luchsinger

The University of Texas at Austin, TX, USA

Abstract

Chemical and molecular systems exist in a world between kinetics and thermodynamics. Engineers of such systems often design them to perform computation solely by following particular kinetic pathways. That is, just like silicon computation, these systems are intentionally designed to run contrary to the natural thermodynamic driving forces of the system. The thermodynamic binding networks (TBN) model is a relatively new model that is particularly well-equipped to investigate this dichotomy between kinetics and thermodynamics. The kinetic TBN model uses reconfiguration energy barriers to inform kinetic pathways. This work shows that deciding if two TBN configurations have a barrier-1 pathway between them is PSPACE-complete. This result comes via a reduction from nondeterministic constraint logic (NCL).

2012 ACM Subject Classification Theory of computation → Models of computation

Keywords and phrases Thermodynamic Binding Networks, Nondeterministic Constraint Logic, NP-complete, PSPACE-complete

Digital Object Identifier 10.4230/LIPIcs.SAND.2022.24

Acknowledgements The author would like to thank the reviewers for their detailed reading and constructive feedback.

1 Introduction and Preliminaries

The thermodynamic binding networks model, first presented in [5], was introduced in order to better study the connection between thermodynamic equilibrium and desired computational pathways. A TBN system, shown in Figure 1(Right), is simply a collection of monomer types which consist of complementary binding sites. Configurations of a TBN system are partitions of the monomers into polymers. A TBN configuration is said to be saturated if the bond count is maximized and said to be stable if it is saturated and the polymer count is maximized. Some recent work has been done on computing properties of stable configurations (i.e., at thermodynamic equilibrium) [3, 6]. Of particular interest to this announcement is the study of energy barriers along kinetic pathways in the TBN model [2] (described by merging and splitting polymers). In other words, how far away from equilibrium do these kinetic paths require the system to be (i.e., how many merges are required before a polymer may be split)? The *1-barrier reachability problem* states: Given two saturated TBN configurations, does there exist a saturated kinetic path between them with barrier at most 1? This announcement shows that this problem is PSPACE-complete by a reduction from constraint logic.

Constraint logic, shown in Figure 1(Left), is a very simple model where orientations are assigned to edges on a graph such that each vertex has minimum total inflow above a certain value (canonically inflow 2). Several questions have been studied in this model, but this announcement focuses on the nondeterministic constraint logic. The nondeterministic constraint logic (NCL) problem (a.k.a. *NCL configuration reachability*) asks if a goal configuration can be reached from an initial configuration through a sequence of edge



© Austin Luchsinger;

licensed under Creative Commons License CC-BY 4.0

1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022).

Editors: James Aspnes and Othon Michail; Article No. 24; pp. 24:1–24:3

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

orientation reversals such that no vertex’s inflow constraint is ever unsatisfied. This problem was shown to be PSPACE-complete in [7]. Constraint logic has been used to show hardness for several problems including various games [7, 4], and motion planning [9, 1].

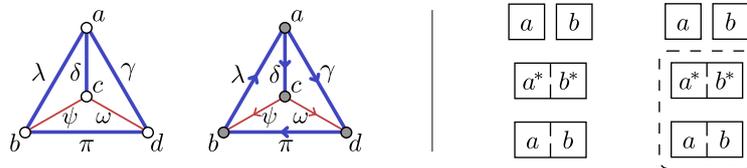


Figure 1 (Left) An example constraint graph $G = (V, E)$ (where $V = \{a, b, c, d\}$ and $E = \{\lambda, \delta, \gamma, \psi, \pi, \omega\}$) and satisfied state G_s . This graph consists of constraint-2 vertices, weight-2 edges (blue) and weight-1 edges (red). It is satisfied because all vertices have inflow of at least two. (Right) An example TBN \mathcal{T} and saturated (and stable) configuration S . The dotted-line box in configuration S indicates that monomers $\{a, b\}$ and $\{a^*, b^*\}$ are part of the same polymer (in this case, satisfying bonds $a-a^*$ and $b-b^*$). S is saturated because all starred domains are bound (bonds are maximized), and it is stable because polymer count is maximized.

2 Complexity Result

Below is a brief explanation of how to transform a constraint graph into a TBN, followed by the outline for the PSPACE-completeness proof.

2.1 Transforming constraint graph into TBN

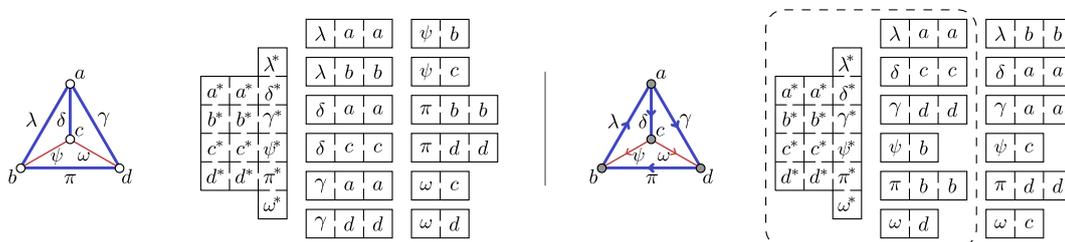
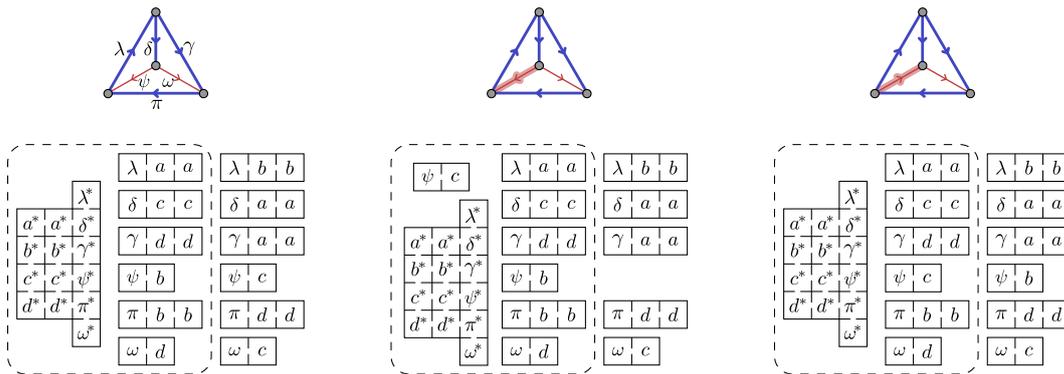


Figure 2 (Left) An example constraint graph G and corresponding TBN \mathcal{T} . (Right) Satisfied state G_s and corresponding saturated configuration S .

Figure 2 illustrates the technique for constructing a TBN given a constraint graph. The primary idea is to construct two “types” of monomers: (1) a *constraint monomer* in the TBN which contains one starred domain per edge and two starred domains per vertex in the constraint graph and (2) two *edge monomers* per edge in the constraint graph (for each edge direction), each with an edge domain and vertex domain(s) dictated by edge weight. It should be noted that this reduction allows for a rather simple proof that a TBN constructed in this way has a saturated configuration with at least $|E| + 1$ polymers if and only if the given constraint graph is satisfiable (both previously known NP-complete problems shown in [3] and [7], respectively).

2.2 Barrier-1 Reachability is PSPACE-complete

Figure 3 is provided as a visual aid to accompany the intuition given in the proof sketch.



■ **Figure 3** An example constraint logic edge flip along with the equivalent TBN merge/split path.

► **Theorem 1.** *The 1-barrier reachability problem is PSPACE-complete.*

A sketch of the proof is as follows: PSPACE-hardness is shown via a reduction from nondeterministic constraint logic reachability. Specifically, there exists a barrier-1 kinetic path between two saturated configurations in the TBN if and only if there exists a valid move sequence between the satisfied constraint logic configurations. The key idea is that a satisfied constraint logic state is represented by a stable TBN configuration with $|E| + 1$ polymers, and a valid edge flip can be simulated by single merge followed by a split operation in the TBN while invalid edge flips require at least two sequential merges. The converse is proven similarly. A simple argument then shows membership in NPSpace which, by Savitch’s theorem [8], means the problem is in PSPACE and thus PSPACE-complete.

— **References** —

- 1 Aaron Becker, Erik D. Demaine, Sándor P. Fekete, Golnaz Habibi, and James McLurkin. Reconfiguring massive particle swarms with limited, global control. In *International Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics*, pages 51–66. Springer, 2013.
- 2 Keenan Breik, Cameron Chalk, David Haley, David Doty, and David Soloveichik. Programming substrate-independent kinetic barriers with thermodynamic binding networks. *IEEE/ACM transactions on computational biology and bioinformatics*, 2019.
- 3 Keenan Breik, Chris Thachuk, Marijn Heule, and David Soloveichik. Computing properties of stable configurations of thermodynamic binding networks. *Theoretical Computer Science*, 785:17–29, 2019.
- 4 Boris De Wilde, Adriaan W. Ter Mors, and Cees Witteveen. Push and rotate: a complete multi-agent pathfinding algorithm. *Journal of Artificial Intelligence Research*, 51:443–492, 2014.
- 5 David Doty, Trent A. Rogers, David Soloveichik, Chris Thachuk, and Damien Woods. Thermodynamic binding networks. In Robert Brijder and Lulu Qian, editors, *DNA Computing and Molecular Programming*, pages 249–266, Cham, 2017. Springer International Publishing.
- 6 David Haley and David Doty. Computing properties of thermodynamic binding networks: An integer programming approach. *arXiv preprint*, 2020. [arXiv:2011.10677](https://arxiv.org/abs/2011.10677).
- 7 Robert A. Hearn and Erik D. Demaine. *Games, puzzles, and computation*. CRC Press, 2009.
- 8 Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.
- 9 Kiril Solovey and Dan Halperin. On the hardness of unlabeled multi-robot motion planning. *The International Journal of Robotics Research*, 35(14):1750–1759, 2016.

